



UNIVERSITÀ DEGLI STUDI DI CATANIA
DIPARTIMENTO DI MATEMATICA E INFORMATICA
CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Sabrina Nicosia

Analisi di reti neurali convoluzionali per la
segmentazione di immagini stradali

RELAZIONE PROGETTO

Docente: Giovanni Maria **FARINELLA**

Anno Accademico 2022 - 2023

Abstract

Nel seguente lavoro viene affrontato il task di segmentazione semantica, in un contesto stradale, andando ad analizzare due reti neurali convoluzionali differenti: *Fully Convolutional Network* e *DeeplabV3*. A seguito dello studio di tali architetture, esse sono state addestrate effettuando un fine-tuning dei modelli pre-allenati forniti dalla libreria Pytorch, disponibile per il linguaggio di programmazione Python, sul dataset Cityscapes. Per ciascun modello sono stati analizzati due backbones differenti, ovvero ResNet-50 e ResNet-101, al fine di comparare le performance dei quattro modelli ottenuti. Questa valutazione è stata condotta durante la fase di training, durante la quale il modello è sottoposto ad un set indipendente da quello di training, ovvero il validation set, su cui vengono calcolate diverse metriche che permettono di identificare il modello che riesca a classificare meglio i pixel di un'immagine, minimizzando gli errori commessi. Una volta identificato il modello finale, è stato sottoposto ad una fase di test in cui vengono calcolate metriche più specifiche che mostrano l'efficacia dell'algoritmo nel predire correttamente ciascuna entità che può essere riscontrata nei vari scenari stradali.

Indice

1	Image Segmentation	3
1.1	Introduzione	3
1.2	Semantic segmentation	4
1.3	Acquisizione dati	5
2	Algoritmi analizzati	7
2.1	Fully Convolutional Networks	7
2.1.1	Downsampling	8
2.1.2	Upsampling	9
2.2	DeepLabV3	9
2.2.1	Atrous convolution	11
2.2.2	Atrous spatial pyramid pooling	11
2.2.3	Upsampling	12
2.3	Addestramento	12
3	Implementazione degli algoritmi	15
3.1	Pre-processing dei dati	16
3.2	Fase di training e validation	18
3.2.1	Metriche	19
4	Benchmarking	21
4.1	FCN	21
4.2	DeepLabV3+	22
4.3	Visualizzazione dei risultati	23
4.4	Modello finale	26
5	Test del modello	27
	Conclusione	29
	Bibliografia	31

Capitolo 1

Image Segmentation

1.1 Introduzione

Con il termine *Image segmentation* [1] si fa riferimento ad una metodologia in cui un’immagine digitale viene suddivisa in diversi sottogruppi, chiamati *image segments*, in modo da ottenere una rappresentazione significativa che consenta di analizzare meglio l’immagine per il task che si intende effettuare. Viene tipicamente utilizzata per individuare i diversi oggetti e confini presenti all’interno di una fotografia. Più precisamente, la segmentazione delle immagini è il processo di assegnare un’etichetta ad ogni pixel in un’immagine in modo tale che i pixel con la stessa etichetta condividano determinate caratteristiche come, ad esempio, colore, intensità o texture. Vi sono diverse tecniche utilizzate per la segmentazione delle immagini, tra cui:

- **Thresholding:** dato un parametro di threshold classifica i pixel come appartenenti al background o foreground verificando se il valore è al di sotto o al di sopra della threshold. Ciò può essere ampliato, in modo da individuare più zone, andando a fornire più valori di threshold, in questo caso si parla di *multi-thresholding*.
- **Edge-based segmentation:** individua i bordi presenti nell’immagine per andare a suddividere le varie regioni o oggetti. I bordi vengono definiti come i confini tra le regioni in un’immagine dove c’è una significativa variazione di intensità o colore.
- **Clustering-based segmentation:** raggruppa i pixel in cluster in base alla loro somiglianza in termini di colore, texture o qualsiasi altra feature che può essere passata all’algoritmo.

La segmentazione di immagini presenta al suo interno diverse sotto-categorie che permettono di adempiere ad uno specifico task da svolgere, tra le quali:

- **Semantic segmentation:** assegna ad ogni pixel dell'immagine un determinato valore, in base al suo significato semantico, che identifica la classe di appartenenza, ovvero un oggetto presente nella scena. Se sono presenti due oggetti della stessa tipologia verrà assegnato loro lo stesso valore.
- **Instance segmentation:** non solo viene assegnata un'etichetta ad ogni pixel dell'immagine, ma viene anche identificata ogni istanza di un oggetto; utile per task che richiedono il conteggio degli oggetti.
- **Panoptic segmentation:** combina le due tecniche precedenti in modo da identificare per ogni pixel la classe di appartenenza e, allo stesso tempo, riconoscere diverse istanze del medesimo oggetto.

1.2 Semantic segmentation

La segmentazione semantica è ampiamente utilizzata in computer vision per assegnare ad ogni pixel di un'immagine un determinato valore che rappresenta tipicamente un oggetto presente al suo interno. Per ottenere tale legame, l'output che si ottiene associa il valore assunto dal pixel ad un valore che rappresenta l'etichetta di classe attraverso l'utilizzo di una mappa di segmentazione. Questo task prende anche il nome di *dense prediction* [2] dato che viene effettuata una predizione per ogni pixel dell'immagine, e ciò permette di ottenere una comprensione dettagliata della scena. A tal scopo, l'immagine viene suddivisa in segmenti significativi che corrispondono a diversi oggetti o regioni di interesse. Ad ogni segmento viene assegnata un'etichetta di classe che indica la categoria di appartenenza; ad esempio "car", "road" o "tree".

In questo campo, gioca un ruolo fondamentale il Deep Learning che ha dimostrato essere estremamente efficace nella risoluzione di questo task; in particolare le Fully Convolutional Neural Network (FCN) [3] hanno segnato un gran passo in avanti dato che sono particolarmente adatte per task che richiedono previsioni a livello di pixel. Come si può intuire dal nome, sono interamente composte da strati convoluzionali, ciò consente loro di acquisire un'immagine in input, di qualsiasi dimensione, e produrre una mappa in output in cui ad ogni pixel viene assegnata la classe in base all'etichetta ottenuta in seguito alla segmentazione effettuata. La principale differenza rispetto alle

tradizionali reti neurali convoluzionali (CNN) [4] è che quest'ultime includono livelli completamente connessi per la classificazione, mentre, le FCN non presentano questa struttura.

Nel tempo sono state introdotte altre architetture di reti neurali appositamente sviluppate per la *semantic segmentation*, come *U-Net*, *SegNet* e *DeepLab*. Queste architetture tengono conto della necessità di mantenere la risoluzione dell'immagine durante il processo di convoluzione e decodifica, preservando così i dettagli dell'immagine.

Tale task è utilizzato, al giorno d'oggi, in diverse applicazioni come la guida autonoma, l'analisi di immagini mediche e nel monitoraggio dei video di sorveglianza, ma anche per distinguere gli oggetti in primo piano dallo sfondo presente in un'immagine. Tutto ciò è reso possibile per merito della segmentazione semantica che fornisce informazioni fondamentali sugli oggetti presenti in un'immagine e sulla loro distribuzione spaziale, consentendo alle macchine di comprendere e interagire meglio con il mondo visivo.

Il seguente lavoro è focalizzato nel campo della guida autonoma al fine di verificare quale sia il modello che riesce a identificare al meglio le varie entità presenti in un contesto stradale.

1.3 Acquisizione dati

Per lo svolgimento del seguente progetto è stato preso in esame un dataset relativo a scenari stradali, chiamato *cityscapes* [5], che presenta al suo interno immagini ad alta risoluzione di strade urbane di diverse città in tutto il mondo. L'acquisizione delle immagini è avvenuta tramite la prospettiva di un'auto in movimento e copre diverse aree urbane, condizioni metereologiche e illuminazioni differenti, così come diversi scenari.

Il dataset è composto da due cartelle: "leftImg8bit" in cui vi sono le immagini originali e "gtfine" che contiene le immagini etichettate, in cui ad ogni pixel è associato un colore rappresentativo della classe a cui è stato assegnato. All'interno di esse vi è una suddivisione in 3 sotto-cartelle che consentono di partizionare le immagini in training, validation e test; al loro intero vi sono ulteriori cartelle che riportano il nome della città in cui sono state catturate, che contengono effettivamente le immagini. Per quanto riguarda il test set, esso non presenta le immagini etichettate, ma sono quelle originali.

Ogni immagine presenta annotazioni a livello di pixel per la segmentazione semantica, ovvero ogni pixel è etichettato con un classe specifica. Tali classi comprendono diversi oggetti e strutture comunemente presenti in scenari urbani, per un totale di 30 classi riconoscibili all'interno del dataset, ovvero: Road, Sidewalk, Building, Wall, Fence, Pole, Traffic Light, Traffic Sign, Vegetation, Terrain, Sky, Person, Rider, Car, Truck, Bus, Train, Motorcycle, Bicycle e Void (Unlabeled regions). Tuttavia, verrà effettuato un raggruppamento al fine di compattare le varie entità che appartengono alla medesima categoria, che sarà mostrato in seguito, nella rispettiva sezione.

Capitolo 2

Algoritmi analizzati

Verranno analizzate due architetture di rete neurale, con differenti componenti e funzionalità, che eseguono il task di segmentazione: *Fully Convolutional Networks* e *DeepLabV3*. Per ciascuna di esse verrà esposta la struttura generale e spiegate le componenti principali al fine di comprenderne il funzionamento.

2.1 Fully Convolutional Networks

Una *Fully Convolutional Networks* (FCN) è un'architettura di rete neurale che ha fatto da pionera per quanto riguarda la segmentazione semantica. È stata introdotta nell'articolo del 2015 intitolato "Fully Convolutional Networks for Semantic Segmentation" [3]. Essa estende le tradizionali reti neurali convoluzionali (CNN) sostituendo gli strati completamente connessi con strati convoluzionali aventi un kernel con dimensione pari a 1x1 in modo da mantenere le informazioni spaziali, consentendo così una previsione pixel per pixel dettagliata. Inoltre, consente di avere una struttura meno complessa, ovvero con meno parametri, che permette di diminuire il tempo computazionale che si avrebbe usando una CNN.

Presenta due componenti principali, chiamate Encoder e Decoder, che hanno il compito di eseguire rispettivamente operazioni di downsampling e upsampling. Nello specifico, il codificatore è responsabile dell'estrazione delle caratteristiche gerarchiche dall'immagine di input, mentre, il decodificatore sovraccampiona queste caratteristiche per generare una mappa di segmentazione a livello di pixel.

2.1.1 Downampling

Nel contesto delle FCN, il downampling si riferisce al processo di riduzione della risoluzione spaziale delle feature maps, ossia l'output di un livello convoluzionale in una rete neurale convoluzionale. Nello specifico, una mappa delle feature è una matrice 3D di valori che rappresenta l'attivazione di un set di filtri appresi su una particolare immagine di input o feature maps da un layer precedente. Ogni filtro è ottimizzato per estrarre una feature o un pattern specifico dall'input, quindi, le feature maps prodotte da un determinato filtro saranno diverse da quelle prodotte da altri filtri nello stesso layer o in layer diversi. I primi livelli acquisiscono feature di basso livello come bordi e angoli, mentre, i layer successivi acquisiscono feature di alto livello come parti di oggetti, texture e forme.

Vi sono due tecniche maggiormente utilizzate per ottenere il downampling: attraverso strati convoluzionali, con un passo (stride) maggiore di 1, o attraverso livelli di pooling.

I layer convoluzionali con uno stride maggiore di 1, che indica la dimensione del passo del filtro convoluzionale mentre si sposta sull'immagine di input, vengono utilizzati per ridurre la risoluzione spaziale delle feature maps. Infatti, con uno stride pari a 2, ad esempio, il filtro si sposterà di 2 pixel alla volta, riducendo le dimensioni dell'output di un fattore pari a 2. Riducendo la risoluzione spaziale delle feature maps, la rete è in grado di acquisire più caratteristiche e modelli globali, riducendo al contempo il calcolo computazionale.

I layer di pooling utilizzano in genere una finestra di dimensioni fisse per calcolare una statistica di riepilogo, ad esempio il valore massimo o medio, su un'area locale delle feature maps. La finestra viene quindi spostata sulla mappa delle funzionalità con un passo fisso, riducendo efficacemente le dimensioni dell'output.

Nelle FCN, il downampling viene in genere eseguito più volte dall'encoder della rete per ridurre gradualmente la risoluzione spaziale delle mappe delle caratteristiche e acquisire più funzionalità globali. L'output del livello di downampling finale viene infine immesso nella parte di decodifica della rete.

2.1.2 Upsampling

L’upsampling, noto anche come ”deconvoluzione” o ”convoluzione trasposta”, è un’operazione cruciale nelle reti neurali convoluzionali (CNN), specialmente nel contesto delle reti completamente convoluzionali (FCN) utilizzate per la segmentazione semantica. Infatti, l’output di una rete neurale è spesso una mappa di caratteristiche a risoluzione inferiore rispetto all’immagine di input, quindi, tale tecnica è necessaria per produrre previsioni in termini di pixel che abbiano le stesse dimensioni spaziali dell’immagine di input.

Uno dei metodo di sovraccampionamento comunemente utilizzati è l’interpolazione bilineare che calcola i nuovi valori dei pixel considerando la media ponderata dei pixel vicini nella mappa delle caratteristiche originale; oppure, può essere applicata un’interpolazione polinomiale, più complessa a livello computazionale ma dal risultato migliore.

Una tecnica più sofisticata è la convoluzione trasposta, che comporta l’applicazione di un’operazione convoluzionale con un filtro apprendibile, chiamato kernel che ha l’effetto di ”diffondere” le informazioni su un’area spaziale più ampia. I parametri del filtro utilizzato vengono appresi durante l’addestramento della rete in modo da ottimizzare il processo di sovraccampionamento. In questo modo, la rete impara come inserire i dettagli mancanti durante l’operazione di sovraccampionamento; inoltre, permette di catturare anche relazioni non lineari.

I livelli di sovraccampionamento vengono generalmente utilizzati alla fine della rete per trasformare le mappe delle caratteristiche a bassa risoluzione in mappe di segmentazione ad alta risoluzione, consentendo alla rete di generare previsioni a livello di pixel.

2.2 DeepLabV3

L’utilizzo delle FCN sulle immagini per task di segmentazione comporta che le feature maps di input diventino più piccole mentre attraversano i livelli convoluzionali e di pooling della rete; da ciò ne deriva una perdita di informazioni che produce un output in cui le previsioni hanno una bassa risoluzione e i confini degli oggetti sono sfocati. Tale difficoltà è stata affrontata dalla famiglia dei modelli DeepLab che utilizzando *Atrous convolutions* e i moduli ASPP (Atrous Spatial Pyramid Pooling). In particolare viene analizzata la versione DeepLabV3 che utilizza una versione migliorata del modulo ASPP,

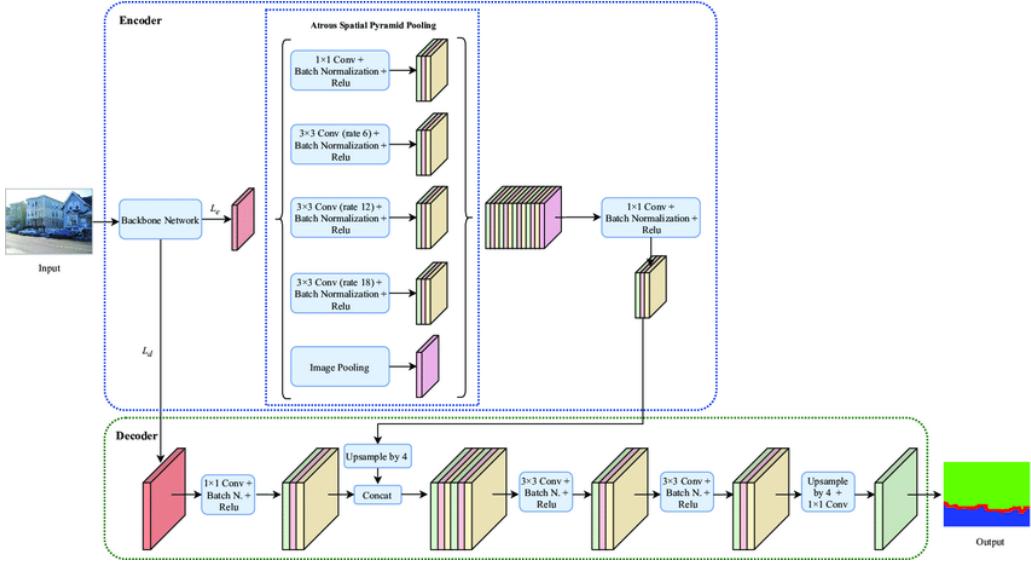


Figura 2.1: Architettura del modello DeepLabV3

nella quale viene introdotta la *batch normalization* e l'estrazione di features a livello di immagine. Inoltre, non fa più uso l'uso del CRF (Conditional Random Field), presente nella prime due versioni.

DeepLabV3, come mostrato in figura 2.1, presenta una struttura encoder-decoder nella quale, il codificatore estrae features di alto livello dall'immagine in input usando come backbone una CNN preaddestrata come, ad esempio, *ResNet*, *MobileNet* o *Xception*. Per controllare le dimensioni delle feature maps, viene utilizzata la convoluzione *atrous* negli ultimi blocchi del backbone. Mentre, il decodificatore perfeziona l'output della segmentazione a livello di pixel eseguendo un upsampling delle features maps, ottenute dal modulo *atrous spatial pyramid pooling* (ASPP), in modo da portarle alla dimensione dell'immagine originale.

DeepLabV3 è ampiamente utilizzato in vari task di computer vision come ad esempio, guida autonoma, analisi di immagini satellitari ed in campo biomedico; task in cui sia il contesto che i dettagli sono fondamentali. Ha dimostrando prestazioni sorprendenti diventando una scelta per molti ricercatori e professionisti in ambito di segmentazione semantica.

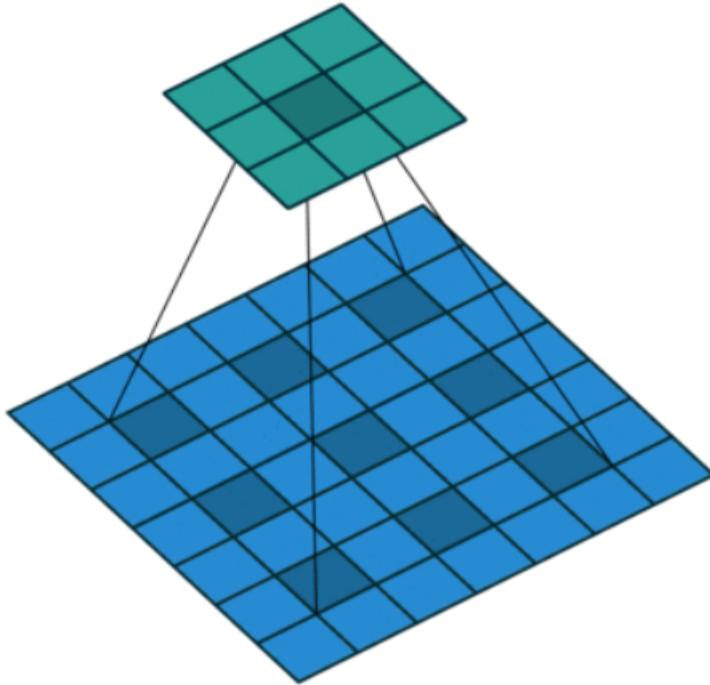


Figura 2.2: Convoluzione di atrous con tasso di dilatazione pari a 2

2.2.1 Atrous convolution

La convoluzione *atrous* viene introdotta in DeepLab come strumento per regolare il campo visivo effettivo della convoluzione, utilizzando un parametro chiamato tasso di *atrous* o di dilatazione (r), senza influire sul calcolo o sul numero di parametri. È simile alla convoluzione tradizionale, tranne per il fatto che il filtro viene ingrandito inserendo zeri tra due valori di filtro successivi lungo ciascuna dimensione spaziale. Vengono creati $r - 1$ buchi tra due valori di filtro consecutivi in ciascuna dimensione spaziale, come mostrato in figura 2.2. Perciò, se il tasso di dilatazione r è pari a 1, si tratta di una convoluzione standard.

2.2.2 Atrous spatial pyramid pooling

L'approccio *atrous*, denominato anche come convoluzioni dilatate, viene inserito dopo il rispettivo backbone del codificatore per acquisire informazioni contestuali multiscala [6]. È tipicamente costituito da rami paralleli, ciascuno dei quali utilizza convoluzioni dilatate con tassi di dilatazione diversi, oltre operazioni di pooling, che elaborano le feature maps in input in modo indipendente. Questi rami consentono al modello di catturare il contesto in

varie scale simultaneamente, consentendo di tener conto di diverse dimensioni e di catturare sia dettagli specifici che un contesto più ampio. In questo modo si ha un miglioramento per quanto riguarda la comprensione dei confini dell’oggetto e dei dettagli presenti nell’immagine, migliorando così la precisione della segmentazione semantica. Inoltre, tali convoluzioni hanno un campo visivo controllato, consentendo al modello di aggregare informazioni su un intervallo spaziale più ampio senza aumentare il numero di parametri.

ASPP incorpora una rappresentazione delle caratteristiche a livello di immagine, che coinvolge il pooling medio globale della mappa delle caratteristiche di input. Questo contesto globale aggregato fornisce informazioni aggiuntive sull’intera immagine che, insieme all’output di ciascun percorso parallelo, consente di ottenere delle feature maps finali ottenute come concatenazione o aggregazione che permettono di catturare il contesto su più scale. Più in dettaglio, nello strato ASPP vengono applicate una convoluzione 1x1 e tre convoluzioni 3x3 con tassi differenti, insieme ad un livello di pooling per ottenere un contesto globale. Tutti i filtri risultanti da ogni ramo vengono concatenati e passati attraverso la convoluzione 1x1 per perfezionare la rappresentazione prima di essere passata al modulo di decodifica.

2.2.3 Upsampling

Le features restituite dell’encoder vengono sottoposte a un upsampling bilineare di un fattore pari a 4, e successivamente concatenate con le corrispondenti caratteristiche di basso livello. Il decodificatore utilizza le cosiddette *skip connections*, connessioni che consentono al decoder di accedere a funzionalità ad alta risoluzione dalle fasi precedenti della rete, consentendo di preservare i dettagli più fini durante il processo di *upsampling*. Viene applicata una convoluzione 1x1 prima di tale concatenazione in modo che il numero di canali possa essere ridotto. Ciò è necessario perché le caratteristiche a basso livello di solito contengono un gran numero di canali che potrebbero prevalere sull’importanza delle features fornite dall’encoder. Dopo la concatenazione, vengono applicate due convoluzioni 3x3 per perfezionare le features, a cui successivamente viene applicato un semplice upsampling bilineare di un fattore 4.

2.3 Addestramento

Nella fase di training dei modelli di *Deep Learning* vi sono alcune componenti cruciali che determinano le prestazioni della rete: ottimizzatore,

funzione di perdita e funzione di attivazione.

L’obiettivo principale dell’ottimizzatore [7] è quello di andare a minimizzare la funzione di perdita regolando i parametri del modello, ovvero pesi e bias, in base al gradiente della funzione di perdita rispetto a tali parametri. Alcuni ottimizzatori comunemente utilizzati includono:

- **Adagrad** (Adaptive Gradient Algorithm): adatta il learning rate per ogni parametro eseguendo passi di aggiornamento più ampi per parametri poco frequenti, mentre, aggiornamenti più piccoli per parametri frequenti.
- **Adam** (Adaptive Moment Estimation): combina la struttura di due metodologie, ossia Adagrad e RMSProp, in modo da adottare i tassi di apprendimento per ciascun parametro in base ai gradienti e agli aggiornamenti passati in maniera efficiente. Usa il concetto di momentum: mantiene una media del decadimento dei gradienti passati che utilizza per accelerare la convergenza nelle direzioni rilevanti.
- **SGD** (Stochastic Gradient Descent): aggiorna i parametri del modello nella direzione opposta al gradiente della funzione di perdita rispetto i parametri. Viene utilizzato come base per diversi modelli.

Un altro componente chiave è rappresentato dalla funzione di perdita, nota come *loss function*. Essa viene utilizzata per misurare quanto le previsioni del modello si discostano dall’etichetta reale, quindi, viene minimizzata al fine di ridurre al massimo la discrepanza tra i due valori durante l’addestramento in modo che il modello apprenda a generare previsioni sempre più accurate.

Nell’ambito della segmentazione semantica la funzione di loss viene applicata a ciascun pixel presente nella mappa di segmentazione fornita in output, con l’obiettivo di ottimizzare la capacità del modello di produrre mappe di segmentazione accurate e precise. A tal scopo posso essere adottate diverse funzioni che variano dalla natura del problema da risolvere, dagli obiettivi dell’addestramento e dalle caratteristiche dei dati di training.

Tipicamente, la funzione di perdita più utilizzata per problemi di classificazione e segmentazione è la *Cross-Entropy loss*. Per un determinato dato di training appartenente alla classe y al quale è stata assegnata una probabilità pari a p , viene calcolata come:

$$\text{Cross - Entropy} = - \sum_i y_i \log(p_i) \quad (2.1)$$

Se il modello assegna una probabilità elevata ad una classe a cui non appartiene quel dato, la funzione crescerà in modo significativo segnalando la necessità di migliorare i parametri del modello al fine di avvicinare le predizioni alle reali etichette dei dati.

La Cross-Entropy Loss è in grado di gestire le problematiche legate allo sbilanciamento delle classi che spesso si verificano nella segmentazione semantica, così come nei problemi di classificazione. Inoltre, non richiede l'implementazione di complessi schemi di pesatura delle classi, semplificando così l'implementazione del modello.

La funzione di attivazione [4] definisce l'output di un neurone a partire dalla combinazione lineare dei pesi e dei valori dei neuroni del layer precedente, in aggiunta al termine di bias. Tali funzioni introducono non linearità nella rete, consentendo di modellare relazioni complesse e non lineari tra input e output. Il suo obiettivo è decidere se, e in che misura, un neurone deve attivarsi in base alla somma pesata dei suoi input. La scelta delle funzione dipende da diversi fattori, tra cui la natura del task ed è fondamentale per il corretto allenamento del modello. Alcune delle funzioni più utilizzate sono:

- **ReLU** (Rectified Linear Unit): ritorna lo stesso valore di input se esso è positivo, altrimenti restituisce 0. La sua formula è, quindi:

$$f(x) = \max(0, x) \quad (2.2)$$

- **Sigmoid**: trasforma l'input x in modo da far sì che il suo valore sia nell'intervallo compreso tra 0 e 1. Viene spesso utilizzata per problemi di classificazione binaria fornendo un output simile alla probabilità di appartenenza ad una classe rispetto che all'altra. È definita come:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.3)$$

- **Softmax**: utilizzata per classificazione multi-classe, preso un vettore x calcola una distribuzione di probabilità esponenziando ciascun valore e normalizzando i risultati in modo che la somma sia 1. La sua formula è:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \quad (2.4)$$

Capitolo 3

Implementazione degli algoritmi

Gli algoritmi illustrati nel capitolo precedente sono stati implementati facendo uso di Pytorch, libreria open-source per il linguaggio Python, ampiamente utilizzata per costruire e addestrare reti neurali.

Il codice è strutturato in un unico file python, chiamato `main.py`, al quale via riga di comando bisogna passare, con l'argomento `--mode`, la modalità che si intende eseguire, ovvero `train`, `test` o `predict`, in base a ciò che si desidera compiere: training del modello, test del modello, oppure ottenere le predizioni su un'immagine. In quest'ultimo caso sarà necessario inserire due argomenti: `--image`, seguito dal path in cui si trova l'immagine di cui si vuole predire la segmentazione, e `--label`, che rappresenta l'immagine con le etichette reali associate ai pixel. L'esecuzione in questa modalità, oltre a calcolare diverse metriche, definite a seguire, nella fase di test del modello, fa comparire un pannello in cui vengono visualizzate tre immagini, ovvero quella originale insieme alla sua immagine di *ground truth*, passate tramite riga di comando, e la predizione ottenuta dal modello finale, ritenuto il migliore.

Di seguito verrà mostrato il pre-processing effettuato, motivando i parametri utilizzati e il rispettivo flusso eseguito, ed analizzata la funzione utilizzata per la fase di addestramento e di validation dei vari modelli, insieme alle metriche adottate.

3.1 Pre-processing dei dati

Il dataset necessario per l’addestramento dei vari modelli è stato scaricato e salvato in una directory chiamata ”cityscape” che presenta al suo interno le cartelle ”leftImg8bit” e ”gtfine”. Per la sua lettura è stato utilizzato il modulo `torchvision.datasets` che fornisce una classe, chiamata Cityscapes, che ne facilità il caricamento e l’utilizzo mediante appositi parametri. In particolare, sono stati settati i seguenti parametri: `root` per specificare il path in cui si trova la directory contenente il dataset; `split` in cui va indicato il rispettivo subset che si vuole utilizzare, a cui è stato passato ’train’; `mode` che indica il livello di granularità o qualità delle annotazioni, che è stato impostato a ”fine”; `target_type` definisce il formato delle immagini di label, a cui è stato passato ”color”. Altri due parametri fondamentali sono `transform`, che permette di applicare delle trasformazioni alle immagini originali e a quelle di label, e `target_transform` per definire operazioni da applicare unicamente per le immagini etichettate.

Per quanto riguarda il parametro `transform` è stato utilizzato per trasformare unicamente le immagini originali, dato che le immagini di ’label’ hanno bisogno di un’elaborazione apposita. Si è fatto uso della classe denominata `transforms.Compose`, che consente di definire una pipeline sequenziale di operazioni da effettuare. Inizialmente è stata inserita la conversione dell’immagine di input in un tensore PyTorch, i cui valori dei pixel vengono automaticamente scalati nell’intervallo [0, 1]. In seguito, è stato eseguito un ridimensionamento dell’immagine in modo da ottenere una size pari a (128,256) usando l’interpolazione bilineare. Infine, è stata applicata una normalizzazione andando a sottrarre la media e dividendo per la deviazione standard in modo da portare i dati di input su una scala comune, migliorando così il processo di training.

Per le trasformazioni da applicare alle immagini di label è stata utilizzata sempre la classe `transforms.Compose`, ma, con operazioni differenti. È stato inserito sempre un ridimensionamento dell’immagine, ma, in questo caso, utilizzando l’interpolazione nearest-neighbor per far sì che non si ottengano colori diversi rispetto a quelli di partenza, dato che ogni colore rappresenta una determinata classe. L’altra operazione consiste nel convertire le immagini dal formato `rgb` in un formato `one-hot-encoding`, richiesto dalle reti neurali. Per far ciò, è stata definita una funzione lambda che applica la funzione `get_one_hot_encoding` ad ogni label. Tale funzione inizialmente mappa la terna `rgb` alla sua classe corrispondente, facendo uso

Group	Classes
flat	road · sidewalk · parking ⁺ · rail track ⁺
human	person [*] · rider [*]
vehicle	car [*] · truck [*] · bus [*] · on rails [*] · motorcycle [*] · bicycle [*] · caravan ⁺⁺ · trailer ⁺⁺
construction	building · wall · fence · guard rail ⁺ · bridge ⁺ · tunnel ⁺
object	pole · pole group ⁺ · traffic sign · traffic light
nature	vegetation · terrain
sky	sky
void	ground ⁺ · dynamic ⁺ · static ⁺

Figura 3.1: Gerarchia delle classi relative agli oggetti identificabili nelle immagini

di una colormap, definita appositamente, che dai 30 colori iniziali definisce unicamente 8 classi: void, flat, construction, object, nature, sky, human e vehicle. Quindi, viene effettuando un raggruppamento che compatta classi simili, seguendo la gerarchia mostrata in figura 3.1. Successivamente viene convertita l'immagine in un tensore pytorch in modo da poter effettuare la chiamata al metodo `one_hot` che restituisce il formato desiderato.

Il training set ottenuto dalla creazione dell'istanza Cityscapes, passando al parametro `split` il valore 'train', è stato suddiviso in due parti in modo da estrarne il validation set. Per far ciò, è stata utilizzata la funzione `random_split` che a partire da un set di dati restituisce due nuovi set non sovrapposti. Oltre al dataset iniziale sono state passate le dimensioni che si desiderano ottenere per i nuovi set, ossia l'0.85 delle dimensioni originali per il training e il restante per il validation. Successivamente, si è fatto uso della classe `DataLoader` che permette, in modo efficiente, di caricare e l'iterare sul dataset che gli viene fornito, andando a suddividerlo in batch. Sono state definite due istanze di tale classe, una per il training set e l'altra per il validation set.

3.2 Fase di training e validation

Per la fase di training e validation dei vari modelli da analizzare è stata definita una funzione, chiamata `train_model` che prende come parametri: il numero di epoche da eseguire, che è stato impostato a 5, l'istanza del `DataLoader` per il training e per il validation set; il modello; la funzione di loss, definita come istanza di `CrossEntropyLoss` presente nel modulo `torch.nn`, l'ottimizzatore, istanza di Adam presente in `torch.optim`, a cui è stato impostato come `learning rate` il valore 0.001; il numero di classi.

I modelli che verranno analizzati, FCN e DeepLabV3, sono definiti facendo uso del modulo `models.segmentation` che supporta due backbone per entrambi, ovvero ResNet50 e ResNet101. Viene impostato, per ogni modello, il parametro `pretrained` col valore "True" in modo tale da avere i parametri inizializzati con i valori ottenuti a seguire del pre-addestramento effettuato su un subset del dataset 'COCO'. Verrà quindi effettuato un fine-tuning dell'intera rete sui 4 modelli distinti, le cui performance saranno illustrate nel capitolo successivo.

All'intero di `train_model` vi è un ciclo che esegue il flusso un numero di volte pari al valore di `num_epochs`, dentro cui vi è un ulteriore ciclo che itera sulle due modalità possibili: `train` e `validation`. Nel primo caso viene impostato il modello in modalità `train` eseguendo il comando `model.train()` in modo tale che si tenga traccia dei parametri allo scopo di calcolare i gradienti durante il passo di backward. Al contrario, `model.eval()` viene chiamato in fase di validation in modo da disabilitare il calcolo dei gradienti.

Al termine di ogni epoca viene eseguito il comando `torch.save` in modo da salvare lo stato del modello, ovvero avere un checkpoint dei parametri del modello in modo che possa essere utilizzato successivamente per altri scopi, come fine-tuning, o semplicemente per testare il modello su altri dati.

Il comando `with torch.set_grad_enabled(mode=='train')` viene utilizzato per abilitare il calcolo dei gradienti delle istruzioni che vengono eseguite al suo interno, nel caso ci si trovi nella modalità di `train`. Internamente è presente un ciclo che itera sui batch del training set, o del validation, che vengono passati al modello che effettua un passo di feedforward, restituendo le rispettive predizioni, sulle quali viene calcolata la funzione di loss. Nel

caso della modalità di train, viene effettuato il passo di backward, nel quale viene utilizzato il comando `loss.backward()` per calcolare i gradienti dei vari parametri del modello rispetto alla funzione di perdita. Dopo che i gradienti sono stati calcolati, viene usato il comando `optimizer.step()` in modo che l'ottimizzatore esegua un passaggio per aggiornare i parametri del modello, ovvero pesi e bias. Infine, come ultimo step del passo di backward, si ha il comando `optimizer.zero_grad()` che azzerà i gradienti di tutti i parametri del modello; necessario prima del successivo passo di forward e backward dato che PyTorch accumula i gradienti per impostazione predefinita invece che sovrascriverli.

3.2.1 Metriche

In seguito ai passi di forward e backward, sono state calcolate diverse metriche per ogni batch, ovvero accuracy, precision, recall e Intersection over Union (IoU). Quest'ultima metrica, comunemente utilizzata per valutare le prestazioni degli algoritmi di image segmentation nel campo della computer vision, misura la somiglianza o la sovrapposizione tra due insiemi. Viene calcolata dividendo l'area in cui le due maschere si sovrappongono per l'area totale coperta da entrambe le maschere, ovvero, l'intersezione tra le zone frutto la loro unione. Viene restituito un valore compreso tra 0 e 1, in cui lo 0 indica una mancata sovrapposizione tra le regioni previste e quelle reali, mentre, 1 indica una corrispondenza perfetta in cui le regioni previste e quelle reali sono identiche. Per la sua implementazione è stata definita una funzione, chiamata `calculate_iou`, che itera su tutte le immagini del batch e, per ogni classe, calcola l'intersezione tra la maschera estratta dall'immagine predetta e la maschera dell'immagine di label, facendo uso del comando `torch.logical_and`. Mentre, viene usato `torch.logical_or` per ottenere l'unione delle due regioni. Facendo il rapporto tra i due valori si ottiene la metrica per singola classe; per ottenere la metrica sull'immagine viene semplicemente fatta una media delle metriche calcolate per classe. Un ulteriore media viene effettuata per ottenere la metrica sul batch. Per quanto riguarda accuracy, precision e recall, vengono utilizzate le metriche implementate dalla libreria `torchmetrics`, su cui sono impostati i seguenti parametri: `task` col valore "multiclass" per indicare che vi sono più classi; `num_classes` a cui è stato passato il valore 8; `average`, settato per precision e recall, che specifica come viene effettuata la media per il calcolo della rispettiva metrica. Viene impostato quest'ultimo parametro col valore "macro" che calcola la metrica per ciascuna classe individualmente e ne estrae la media, non ponderata, di tutte le classi.

Le metriche non presenti in `torchmetrics` vengono calcolate servendosi della classe `AverageValueMeter()`, il cui costruttore resetta le due variabili `sum` e `num`, usate rispettivamente per sommare i valori che gli vengono forniti e per tenere traccia del numero di valori passati. Presenta il metodo `add` che permette di passare un valore, che corrisponde ad una media, che viene moltiplicato per il parametro `num`, che indica il numero di istanze che hanno contribuito alla media, il tutto viene sommato al cumulato dei valori precedentemente passati. Infine, il metodo `compute` permette di ottenere la media, calcolata come rapporto tra `sum` e `num`.

Le metriche vengono calcolate, quindi, per ogni batch, poi passate al metodo `add`, infine, viene chiamato il metodo `compute` per ottenere i valori per la rispettiva epoca. Inoltre, per calcolare la media di ogni metrica su tutte le epoche, alla fine di ogni epoca si tiene traccia del valore ottenuto su un’istanza distinta della classe `AverageValueMeter`. In questo modo, chiamando il metodo `compute` su tutte le istanze di `AverageValueMeter`, al termine di tutte le epoche, si otterrà la media desiderata per ogni metrica.

Dato che le varie metriche devono essere calcolate sia per il training set, che per il validation set, è stato definito un dizionario, chiamato `metrics_dict` che presenta al suo interno due chiavi: `train` e `val`. Come valore presentano entrambe due dizionari che hanno tante chiavi quante sono le metriche da calcolare. In questo modo per salvare le metriche corrispondenti alla specifica modalità in cui ci si trova basterà eseguire il comando `metrics_dict [mode]`. Lo stesso è stato effettuato per salvare la media delle metriche calcolate per ogni epoca, definendo il dizionario `avg_metrics_dict`.

Capitolo 4

Benchmarking

Al fine di adottare l’architettura che dia i risultati migliori per il task di segmentazione, sono state analizzate le metriche calcolate durante la fase di addestramento, sui diversi backbones disponibili per ciascun modello.

4.1 FCN

In tabella 4.1 sono mostrate le varie metriche, ciascuna delle quali ottenuta effettuando una media dei valori calcolati per ogni epoca, per i due backbones. Evince immediatamente la similarità dei valori ottenuti per entrambe le architetture, sia sul training set, che sul validation set. Si può notare, inoltre, come i due modelli sembrano generalizzare piuttosto bene. Tuttavia, andando a considerare i valori ottenuti sul validation set nelle singole epoche è emerso che l’architettura ResNet-50 raggiunge le prestazioni migliore nell’epoca numero quattro, peggiorando quindi nell’ultima. Mentre, i valori sul training set nell’ultima epoca rimangono pressoché simili. Ciò non è un buon segno, dato che, se la funzione di loss aumenta nel validation set, si può verificare un caso di overfitting in cui il modello non riesce a generalizzare bene. In questo caso, andrebbe aggiustato di conseguenza il learning rate; tuttavia, date le poche risorse computazionali, ci si limiterà a comparare i modelli unicamente sulle prime 5 epoche eseguite.

Per quanto riguarda la ResNet-101, la funzione di loss decresce su ogni epoca, sia sul training che sul validation set, segno del fatto che la rete sta apprendendo correttamente senza adattarsi troppo ai dati di training. Perciò, in questo caso, sarebbe preferibile adottare questa architettura, anche se, essendo più complessa, ovvero avendo più layer nascosti rispetto alla pre-

Backbone	Set	Loss	Accuracy	Precision	Recall	IoU
ResNet-50	Training	0.43	0.86	0.74	0.69	0.51
	Validation	0.45	0.85	0.76	0.68	0.50
ResNet-101	Traning	0.43	0.86	0.76	0.69	0.50
	Validation	0.44	0.85	0.75	0.69	0.51

Tabella 4.1: Metriche ottenute durante l’addestramento del modello FCN con diversi backbones

Backbone	Set	Loss	Accuracy	Precision	Recall	IoU
ResNet-50	Training	0.44	0.86	0.75	0.69	0.50
	Validation	0.51	0.83	0.75	0.67	0.49
ResNet-101	Traning	0.45	0.85	0.75	0.68	0.50
	Validation	0.56	0.83	0.73	0.67	0.48

Tabella 4.2: Metriche ottenute durante l’addestramento del modello DeepLabV3 con diversi backbones

cedente, presenta più parametri da ottimizzare, e richiederebbe un training maggiore dato che la loss stava continuando a decrescere.

4.2 DeepLabV3+

La stessa analisi è stata effettuata per il modello DeepLabV3, i cui risultati sono riportati in tabella 4.2.

Per quanto riguarda il backbone ResNet-50, presenta lo stesso problema citato per il modello precedente, ovvero nelle ultime due epoche la loss sul validation set aumenta, come si evince dal valore ottenuto, ancor più alto che nel caso precedente. Ciò porta ad escludere questa architettura anche per questo modello.

Mentre, sebbene la funzione di loss sia ancor maggiore per il backbone ResNet-101, trattandosi di una media, è influenzata dai valori iniziali, che, in questo caso, sono stati più alti. Infatti, un’analisi più accurata ha mostrato che, considerando l’ultima epoca di questo backbone, che ha riportato un valore di loss pari a 0.36, per il trainin set, e 0.38, per il validation set, si comporta decisamente meglio rispetto ai restanti 3 modelli, infatti, ottiene valori migliori comparati con le epoche migliori di tutti i modelli analizzati. Ciò porta a prediligere quest’ultimo modello, ma, per dimostrare la sua

bontà, verranno mostrati alcuni esempi visivi delle predizioni che produce, comparate con quelle degli altri modelli.

4.3 Visualizzazione dei risultati

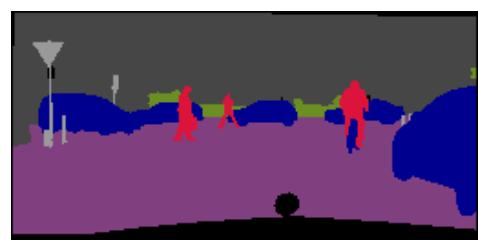
In supporto alle metriche calcolate, vengono mostrati due esempi di predizioni ottenute da ciascun modello, in corrispondenza dell'ultima epoca in cui è stato allenato, al fine di compararne i risultati e verificare quale, nel complesso, si avvicina di più alla rispettiva immagine di *label*, ovvero riesca ad identificare meglio gli oggetti presenti nella scena.

Come primo esempio si guardi la figura 4.1, che riporta, nella prima riga, l'immagine originale con la sua rispettiva *label*, mentre, le altre due righe contengono le predizioni dei quattro modelli analizzati. Si può notare immediatamente come la predizione con Deeplab usando come backbone ResNet-50 presenta una vasta zona classificata erroneamente di nero. Ciò è dovuto al fatto che tale modello nelle ultime due epoch è andato a peggiorare, ovvero la loss sul validation set, invece di continuare a decrescere, è aumentata. Ciò, come si può vedere, porta ad avere predizioni errate nel validation set, segno del fatto che il modello non riesce a generalizzare. Per quanto riguarda gli altri modelli, in questo caso, hanno predetto tutti piuttosto bene, anche se, nessuno riesce ad identificare i segnali stradali, rappresentati col colore grigio chiaro. Inoltre, FCN con backbone ResNet101 presenta un po' di rumore nella zona col grigio scuro, che identifica un edificio, confusa con l'etichetta associata alle autovetture.

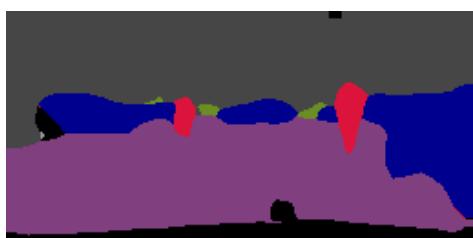
Il secondo esempio, che va indubbiamente a favore del modello Deeplab con backbone ResNet101, è mostrato in figura 4.2. I due modelli ottenuti con la FCN, sebbene riescano ad identificare nel complesso gli oggetti presenti, commettono diversi errori: il primo modello ingloba nei pixel associati con l'etichetta 'human', anche un pezzo di autovettura, che non è in grado di riconoscere; mentre, il secondo non identifica affatto che ci siano dei pedoni. Il quarto modello rappresenta un buon compromesso dato che è in grado di identificare piuttosto bene tutte le zone presenti, tuttavia, non riesce ad individuare particolari più fini, come i pali presenti in questo esempio. Ciò può essere dovuto ai pochi esempi che si hanno per questa classe, dato che non è sempre presente, tuttavia, questo aspetto verrà analizzato meglio a seguire.



(a) Immagine originale



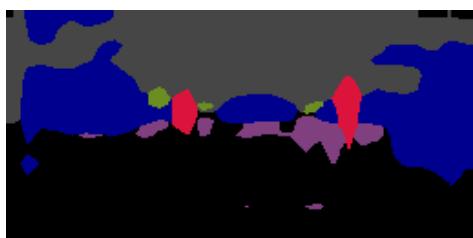
(b) Immagine di *label*



(c) Predizione con FCN e ResNet50



(d) Predizione con FCN e ResNet101



(e) Predizione con Deeplab e ResNet50

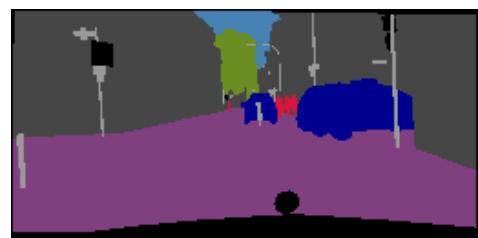


(f) Predizione con Deeplab e ResNet101

Figura 4.1



(a) Immagine originale



(b) Immagine di *label*



(c) Predizione con FCN e ResNet50



(d) Predizione con FCN e ResNet101



(e) Predizione con Deeplab e ResNet50



(f) Predizione con Deeplab e ResNet101

Figura 4.2

Backbone	Set	Loss	Accuracy	Precision	Recall	IoU
ResNet-101	Training	0.37	0.87	0.78	0.73	0.54
	Validation	0.39	0.87	0.79	0.71	0.53

Tabella 4.3: Metriche ottenute durante la quinta epoca dell’addestramento del modello DeepLabV3 con backbone ResNet-101

4.4 Modello finale

Dalle considerazioni precedentemente effettuate emerge che il modello che, nel complesso, si comporta meglio è DeepLab con backbone ResNet-101. In tabella 4.3 vengono mostrate le prestazioni ottenute sulla quinta, ed ultima, epoca in cui è stato addestrato. Tali performance sono risultate le migliori anche graficamente portando quindi alla seguente decisione finale. Nel successivo capitolo verrà sottoposto ad un nuovo set di immagini, ancora non processate, in modo tale da testarlo ed analizzarne alcuni aspetti più nel dettaglio.

Capitolo 5

Test del modello

Una volta individuato il modello con le performance migliori, è stato sottoposto ad un nuovo set di dati che possiede le *label* associate, ovvero il validation set. Ciò viene effettuando eseguendo il codice, da riga di comando, passando l'argomento `--mode` seguito dalla stringa `test`, che indica la modalità di test. Al momento il codice è strutturato in modo da effettuare il test unicamente sul modello DeepLabV3 con backbone ResNet-101.

Il validation set presenta 500 immagini che sono state date in pasto all'algoritmo in modo tale da ricavarne delle metriche che permettano di stabilire se effettivamente il modello riesce a identificare correttamente le varie classi presenti nelle immagini. In questa fase, oltre alle metriche definite durante l'addestramento, ovvero loss, accuracy, precision e recall, la metrica IoU è stata calcolata separatamente per ogni classe in modo tale da identificare quali sono le classi che riesce a classificare meglio, rispetto a quelle che non coglie del tutto. Le prime quattro metriche sono riportate in tabella 5.1, la quale mostra valori piuttosto buoni, nel complesso, che indicano che l'85% dei pixel, in media, vengono classificati correttamente. Tuttavia, tale metrica viene calcolata nel complesso, non andando a considerare le singole classi, perciò in tabella 5.2 viene mostrata la metrica IoU per ogni classe, essendo più significativa. Si può notare come vi siano due classi, *Object* e *Human*, che presentano valori decisamente troppo bassi, infatti si tratta di classi che sono meno presenti e con pattern molto specifici, che l'algoritmo non è riuscito a catturare del tutto bene, come è stato mostrato anche nel capitolo precedente. Al contrario, la classe *Flat*, che identifica porzioni come la strada, presenta un valore molto alto, segno del fatto che la porzione di pixel classificati con questa classe si avvicina molto ai pixel che possiedono tale etichetta. Le restanti classi presentano, allo stesso tempo, valori piuttosto buoni.

Loss	Accuracy	Precision	Recall
0.47	0.85	0.77	0.70

Tabella 5.1: Metriche ottenute durante la fase di test del modello DeeplabV3 con backbone ResNet-101

Void	Flat	Construction	Object	Nature	Sky	Human	Vehicle
0.58	0.89	0.65	0.06	0.66	0.57	0.13	0.58

Tabella 5.2: IoU score calcolato per ogni classe durante la fase di test del modello DeeplabV3 con backbone ResNet-101

Nel complesso, ciò che si dovrebbe andare a migliorare è il riconoscimento dei dettagli più fini, ovvero delle classi *Object* e *Human*, per i quali sarebbe necessario eseguire un numero maggiore di epoche durante il training in modo tale da migliorarne il riconoscimento. Infatti, considerando il numero esiguo di epoche con cui è stato allenato il modello in esame, i risultati ottenuti sono abbastanza buoni.

Conclusione

Dallo studio della libreria Pytorch, insieme alle componenti e funzionalità necessarie per l’addestramento dei modelli proposti, *Fully Convolutional Network* e *DeeplabV3*, è stato possibile effettuare il fine-tuning dei modelli pre-addestrati che Pytorch offre per il task di segmentazione semantica. Nello specifico, sono disponibili due backbones per ciascun modello, i cui pesi sono stati caricati ed utilizzati come punto di partenza per il riaddestramento sul dataset Cityscapes. I pesi sono stati ricalcolati sull’intera rete dato che si vuole analizzare un contesto prettamente stradale, parecchio differente rispetto al dataset COCO, sul quale sono stati pre-allenati i modelli, che presenta immagini che appartengono a diversi domini.

Durante la fase di training, oltre all’aggiornamento dei parametri, viene eseguita una fase di validation in cui il modello viene sottoposto ad un insieme di immagini, indipendente dal training set, che permette di farsi un’idea delle capacità del modello di predire la classe corrispondente dei pixel di immagini che non ha analizzato in precedenza. Perciò, le metriche definite all’interno della funzione che si occupa di eseguire il training del modello, vengono calcolate anche nella fase di validation in modo da identificare il modello che riesce a minimizzare gli errori commessi. Le metriche sui due sottoinsieme, calcolate per ogni epoca, sono state fondamentali per valutare quale sia il modello migliore e, dall’analisi svolta, è emerso che DeepLabV3 con backbone ResNet-101 si comporta meglio, ottenendo un valore di loss, sul validation set, inferiore rispetto ai restanti modelli. Per accettare ciò, si è fatto uso della rappresentazione grafica delle predizioni ottenute coi vari modelli, che ha fatto emergere la bontà di tale modello.

Infine, il modello migliore è stato sottoposto ad una fase di test, nella quale è stata calcolata la metrica IoU per ogni classe da discriminare, in modo tale da far emergere quali entità vengono riconosciute meglio. Tale analisi ha rivelato che, nel complesso, le classi vengono identificate correttamente, ad eccezione di due: *Object* e *Human*. Ciò è dovuto al fatto che rappresentano

oggetti più fini e meno presenti rispetto alle restanti entità, perciò, la loro identificazione richiede una fase di training maggiore, quindi un numero di epoche superiore a 5. Quindi, come possibile sviluppo futuro si potrebbe aumentare il numero di epoche e vedere se il modello riesce a identificare meglio le due classi meno rappresentate, senza andare a compromettere le performance delle restanti classi. Inoltre, per evitare problemi di overfitting, posso essere adottate tecniche come il Dropout o la Data Augmentation.

Bibliografia

- [1] Wikipedia contributors. Image segmentation — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Image_segmentation&oldid=1176878667, 2023. [Online; accessed 20-November-2023].
- [2] Jeremy Jordan. An overview of semantic image segmentation. <https://www.jeremyjordan.me/semantic-segmentation/>, 2023. [Online; accessed 20-November-2023].
- [3] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.
- [4] Shadman Sakib, Nazib Ahmed, Ahmed Jawad Kabir, and Hridon Ahmed. An overview of convolutional neural network: Its architecture and applications. *Preprints*, February 2019.
- [5] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [6] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.
- [7] Musstafa. Optimizers in deep learning. <https://medium.com/mlearning-ai/optimizers-in-deep-learning-7bf81fed78a0>, 2022. [Online; accessed 30-November-2023].