



---

# *Relazione progetto CV*

## *“SSAI”*

---

Alunno:

Fiorito Aldo 1000038099

Professori:

Sebastiano Battiato

Francesco Guarnera

Corso di laurea magistrale LM-18

Computer Vision

## Sommario

Introduzione	3
Problema affrontati nel progetto: ISS	3
Reti CNN	6
Librerie usate e configurazione sistema	7
Svolgimento e struttura codice	8
Gestione delle immagini	11
Valutazioni e metriche	16
Benchmarking	20
Conclusione e considerazioni	25

Repository URL : <https://github.com/TheNor01/SSAI>

## Introduzione

La seguente relazione mira a spiegare il lavoro svolto su un particolare task nel mondo della Computer vision. Quest'ultimo si occupa della creazione di sistemi che possono interpretare, analizzare e comprendere le informazioni visive dal mondo reale.

Tra gli obbiettivi più comuni della CV troviamo:

1. Il riconoscimento degli oggetti
2. La segmentazione semantica delle immagini
3. Riconoscimento facciale

Al fine di risolvere questi problemi, la tecnologia e la scienza hanno trovato diversi approcci e risoluzioni. Tra i meno recenti troviamo metodi basati sui filtri per identificare una particolare forma target, oppure, metodi più incentrati sul matching di template base con l'immagine interessata. Tuttavia, con l'avvento del Machine Learning e del consecutivo Deep Learning, lo stato attuale dell'arte trova diverse novità, basate maggiormente su approcci più classici (come il kmeans per identificare pixel appartenenti allo stesso oggetto), oppure altre tecniche basate sulle reti convoluzionali (CNN), ricorrenti (RNN) e generative (GAN)

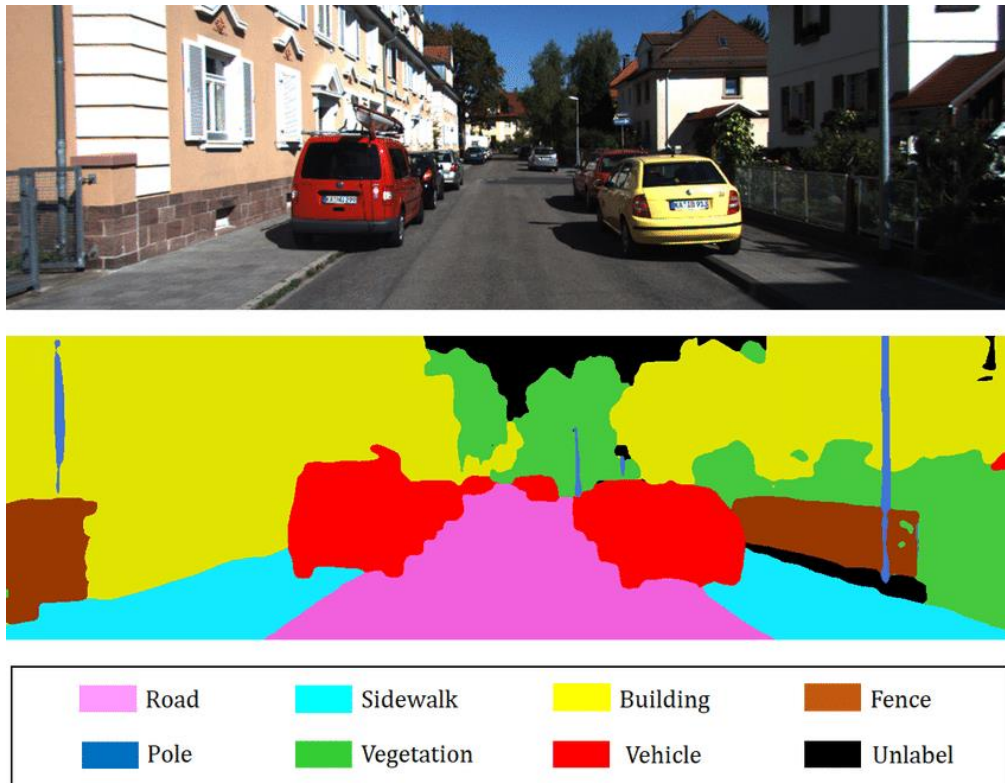
## Problema affrontati nel progetto: ISS

Nel seguente progetto si è scelto, in particolar modo, di approfondire il task della segmentazione semantica (semantic segmentation image SSI) di immagini e di comparare tra di loro diverse implementazioni di CNN.

Per capire meglio il problema, possiamo definire il problema in questo modo:

*"La segmentazione delle immagini è un processo nell'ambito della computer vision che consiste nel suddividere un'immagine in diverse regioni o segmenti, in modo da rappresentare l'immagine in qualcosa di più significativo e più facile da analizzare.*

*L'obiettivo principale della segmentazione delle immagini è dividere l'immagine in regioni omogenee in termini di colore, intensità, texture o altre proprietà”*



*Figura 1 Esempio image segmentation*

Ad esempio, presa l'immagine di sopra e presi un set di entità semantiche, possiamo identificare e associare ad ogni regione (gruppo di pixel) una etichetta. Questa etichetta ha un valore molto importante in termini semantici per un computer.

Si pensi ad un sistema di guida autonoma. L'intelligenza artificiale deve saper distinguere tra una entità strada ed una entità persona. Essi devono avere un valore diverso al fine di poter prendere diverse decisioni diverse ogni volta che il sistema ne identifichi una.

Il numero di etichette, che da ora chiameremo labels/classi, è totalmente arbitrario, o comunque limitato dal significato semantico che un essere umano potrebbe applicare guardando l'immagine.

Per il nostro progetto si è scelto di usare un dataset relativo a scene di strada urbana, fornito da <https://www.cityscapes-dataset.com/dataset-overview/>. Questo è stato formato estraendo frame da vari video girati in 50 città, in diverse stagioni e orari.

The following packages are available for download. Please download only those packages that you need for your immediate research.





	<a href="#">gtFine_trainvaltest.zip (241MB) [md5]</a> fine annotations for train and val sets (3475 annotated images) and dummy annotations (ignore regions) for the test set (1525 images)
	<a href="#">gtCoarse.zip (1.3GB) [md5]</a> coarse annotations for train and val set (3475 annotated images) and train_extra (19998 annotated images)
	<a href="#">leftImg8bit_trainvaltest.zip (11GB) [md5]</a> left 8-bit images – train, val, and test sets (5000 images)
	<a href="#">leftImg8bit_trainextra.zip (44GB) [md5]</a> left 8-bit images – trainextra set (19998 images, note that the image "troisdorf_000000_000073_leftImg8bit.png" is corrupt/black)

Figura 2 Dataset image segmentazion

Il dataset propone la possibilità di riconoscere fino a 30/31 classi all'interno di una immagine, con la possibilità di aggregare più entità semantiche qualora i risultati non dovrebbero essere del tutto chiari.

Classi proposte

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

\* Single instance annotations are available. However, if the boundary between such instances cannot be clearly seen, the whole crowd/group is labeled together and annotated as group, e.g. car group.  
+ This label is not included in any evaluation and treated as void (or in the case of *license plate* as the vehicle mounted on).

Figura 3 Classi Dataset

Per il nostro task sono stati scaricati i file chiamati

- Leftimg8bit\_trainvaltest.zip (11GB)
- gtFine\_trainvaltest (241MB)

Queste due cartelle rappresenteranno le nostre immagini, su cui alleneremo le reti neurali, insieme alle loro segmentazioni già costruite

## Reti CNN

Come anticipato prima, tra gli approcci esistenti al giorno d'oggi, i più promettenti ed i più avanzati sono sicuramente le reti neurali.

Esaminando lo stato dell'arte attuale della ricerca si è deciso di provare a risolvere il task della SSI con le seguenti implementazioni di CNN. Ognuna di loro offre una diversa sfaccettatura

1. **UNET**, creata originariamente per scopo medico. Non implementa particolari caratteristiche di riferimento  
(<https://arxiv.org/pdf/1505.04597.pdf>)
2. **RES-Unet**, implementa la possibilità di collegare due layer attraverso dei "residui" e di attivare layer particolari in base a questi residui. Si basa sul principio di un'altra rete, la RESNET  
(<https://arxiv.org/pdf/1904.00592.pdf>)
3. **DEEPLAB**, implementa delle convoluzioni particolare chiamate "*Atrous Convolution*", in cui, si espande la dimensione del kernel secondo un parametro di "dilatazione", durante le convoluzioni.  
(<https://arxiv.org/pdf/1606.00915v2.pdf>)

NB. Dai seguenti PDF si è preso di riferimento solamente alla struttura della CNN e non l'intera pipeline implementativa.

## Librerie usate e configurazione sistema

Il seguente progetto è stato realizzato interamente usando il linguaggio python e le relative librerie utili ai task di deep learning e analisi dei dati messi a disposizione. Per dare una idea dei principali package utilizzati

- Python3.9.8
- Sklearn metrics
- matplotlib
- Numpy
- Scipy
- Cv2
- Pytorch
- Torchvision
- Tensorboard
- Opencv-python

NB.

La configurazione di sistema utilizzata per il task non prevede l'utilizzo di pytorch CUDA poiché non si dispone di schede video NVIDIA

Dunque, ogni tensore calcolato durante l'addestramento della CNN è disponibile solo su device cpu.

Nello specifico il modello di cpu utilizzato è un AMD Ryzen 3600 con 6 core fisici, 12T, 35 MB di cache e con un cpu boost 4,2 GHz

La ram massima del sistema è di 16 GB DDR4

## Svolgimento e struttura codice

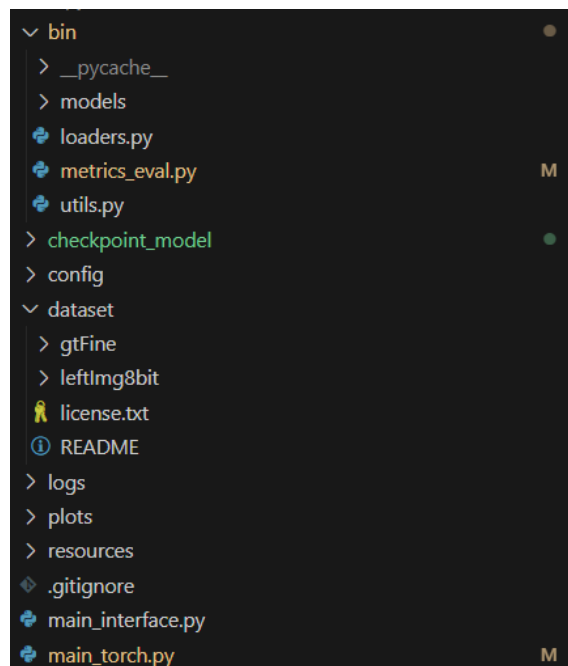


Figura 4 Struttura codice

All'interno di *bin* troviamo tutti i modelli utilizzati all'interno del codice, oltre che gli script di appoggio per costruire i dataset, caricare i dati dal FS e trainare/valutare la rete.

Inoltre, sono presenti codici utili al fine di gestire correttamente le segmentazioni.

Dentro la cartella *models* è possibile trovare le classi delle reti CNN

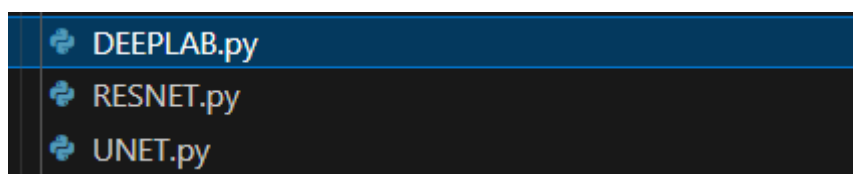


Figura 5 Moduli reti

Le ultime due sono implementate from scratch, dunque è possibile ispezionare modificare i layer a piacimento.

Per quanto riguarda la rete deeplab, è stata adottata una implementazione già costruita, messa a disposizione da torchvision, recuperabile da questa pagina: <https://pytorch.org/vision/stable/models.html>. Oltre a caricare il modello sarà necessario caricare i pesi di default della rete.

Per adattare la rete al nostro problema è necessario aggiungere uno step di



adattamento nell'ultimo livello, questo per il numero di classi presenti in input.

```
model.classifier[4] = nn.Conv2d(256, num_classes, 1) # decoder to -->adapt last  
layer to our classes
```

Su logs e plots, saranno disponibili metriche utili al fine di capire l'andamento della rete e la sua valutazione.

In particolar modo, su logs saranno presenti i valori della loss usata durante la procedura di training. Questi file, insieme ai loro metadati saranno visibile su tensorboard, una interfaccia utile a mostrare grafici.

Nell'immagine sotto è possibile vedere un esempio dei valori della loss, durante la procedura di training.

Opzionalmente, è possibile plottare anche la loss durante la fase di testing.

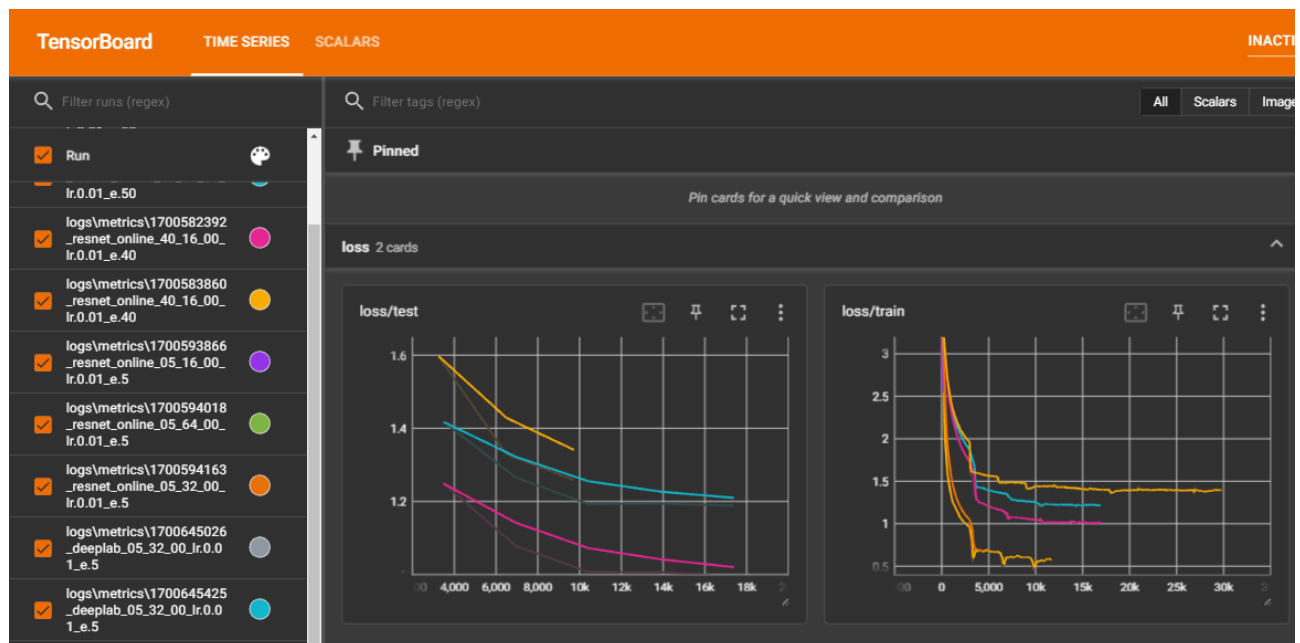


Figura 6 TensorBoard

Su config/labels.py sono disponibili le labels con cui andremo a indentificare le entità all'interno dell'immagine. Queste sono prese dal repository di cityscapes al seguente link:

<https://github.com/mcordts/cityscapesScripts/blob/master/cityscapesscripts/helpers/labels.py>

Ogni LABEL è identificata da vari ID (quelli relativi alla label più a basso livello e quelli più “astratti”), oltre che ad un valore RGB. Quest’ultimo è il colore con cui viene identificata l’entità all’interno della ground truth fornita del dataset

```
Label( 'parking'      , 9 ,      255 , 'ground'      , 1 , False , True  , (250,170,160) ),
Label( 'rail track'   , 10 ,     255 , 'ground'      , 1 , False , True  , (230,150,140) ),
Label( 'building'     , 11 ,       2 , 'construction' , 2 , False , False , ( 70, 70, 70) ),
Label( 'wall'         , 12 ,       3 , 'construction' , 2 , False , False , (102,102,156) ),
Label( 'fence'        , 13 ,       4 , 'construction' , 2 , False , False , (190,153,153) ),
```

Figura 7 Labels

All’interno della cartella *gtFine*, troviamo le nostre immagini a colori, con la denominazione “*gtFine\_color.png*”.



Figura 8 Ground Truth

Infine, nella cartella “*checkpoint\_model*”, troviamo i pesi dei modelli salvati dopo la procedura di training. Questo è fatto al fine di ripristinare i modelli senza dover rieseguire nuovamente la pipeline intera.

### *Gestione delle immagini*

Le immagini collezionate vengono, prima di essere passate alla rete, ritagliate ad una grandezza predefinita. Più questa sarà grande, più la rete sarà capace di cogliere le corrette entità all'interno della immagine.

Il training set originale contiene immagini con una risoluzione pari a 2048x1024 pixel, con una profondità di colore pari a 24bit.

Successivamente, vengono letti rispettivamente i due path (immagine e label immagine) con cv2 e riconvertiti in scala RGB (questo poiché internamente opencv utilizza uno sbaglio di colore diverso).

Il nostro interesse è quello di usare il colore RGB in quando la terna di colori delle label è espressa in tale spazio di colori.

```
image = cv2.imread(img_path, cv2.IMREAD_COLOR)
image = cv2.resize(image, (self.size, self.size))
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB).astype('float32')
image = image / 255.0
mask = cv2.imread(lbl_path, cv2.IMREAD_COLOR)
mask = cv2.resize(mask, (self.size, self.size))
mask = cv2.cvtColor(mask, cv2.COLOR_BGR2RGB).astype('float32')
```

Successivamente, adottiamo una trasformazione della nostra immagine label, da adesso in poi chiamata maschera.

Per spiegare meglio la trasformazione è bene illustrare quale sarà l'approccio generale che le 3 CNN implementeranno.

Immaginiamo che sovrapposta alla nostra immagine originale ci sia una maschera di interi, dove ogni intero diverso rappresenta una diversa entità.

Regioni della stessa entità avranno pixel uguale.

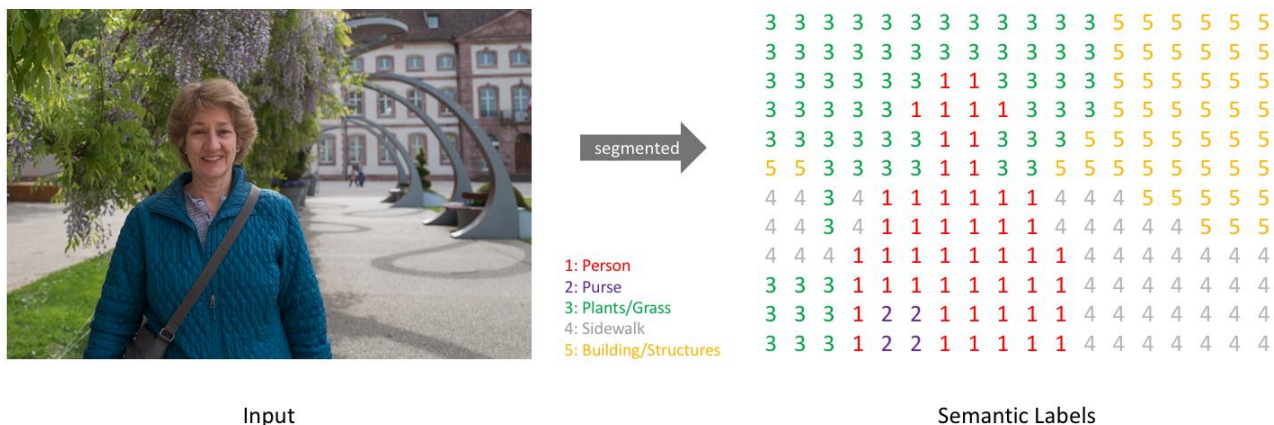


Figura 9 Logica SSI

La nostra rete dovrà riuscire a tirare fuori questi valori interi e comporre la nostra immagine.

Gli output della rete CNN sarà del formato  $(1 \times H \times W \times \text{classes})$ , dove 1 sarà il numero del batch (possiamo ometterlo) e classes saranno i canali in cui verranno distribuiti gli interi.

In sostanza ogni entità avrà un canale dedicato.

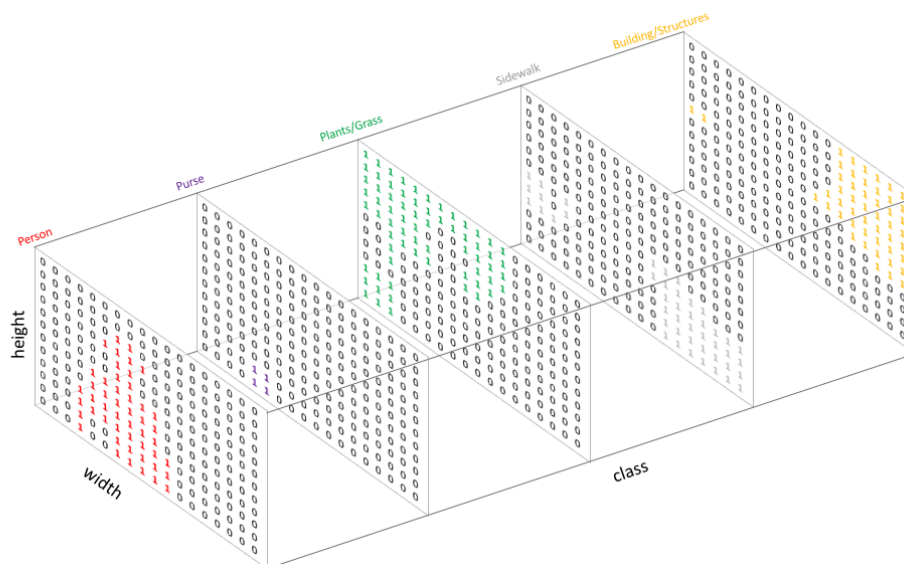


Figura 10 Channel Immagine modello

Per poter ricostruire l'immagine originale dovremmo prendere i massimi di ogni canale e comporre nuovamente l'immagine. Come?

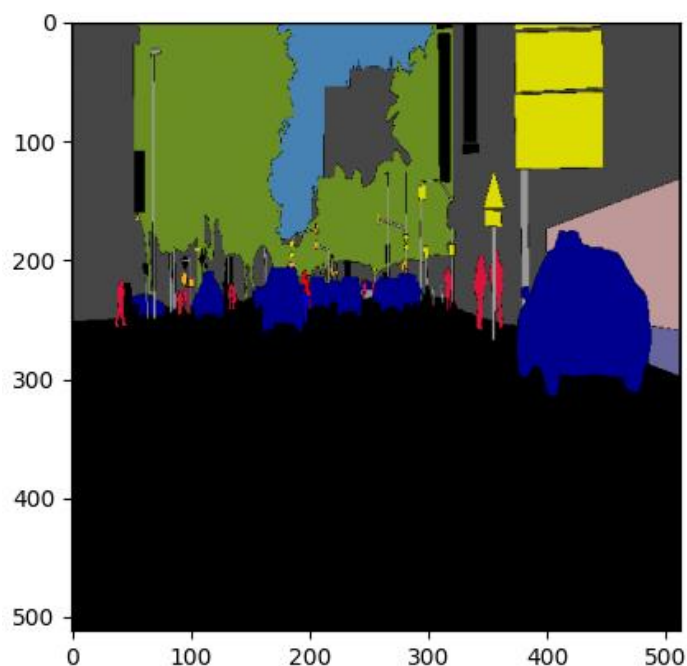
Effettuando l'operazione inverso all'encoding verso l'intero e ricostruire il valore RGB della classe.

Ecco il principio su cui si baserà la risoluzione del task di SSI

Dunque, una volta lette le due immagini attraverso opencv, passeremo la nostra immagine y (ground truth) attraverso la funzione di encoding. Questa, come parametri prenderà l'immagine di riferimento, le chiavi (id) delle classi e i loro valori di riferimento.

L'output, come detto prima, sarà una matrice (H x W) di interi.

Ad esempio, presa un'immagine di test, e applicando la funzione di encoding



*Figura 11 Immagine label visualizzata*

## ENCODING

```
def get_label_mask(self, mask, class_values, label_colors_list):
    label_mask = np.zeros((mask.shape[0], mask.shape[1]), dtype=np.uint8)
    #3d zero

    for value in class_values:
        for i, label in enumerate(label_colors_list):
            if value == label_colors_list.index(label):
                label = np.array(label)
```

```

        label_mask[np.where(np.all(mask == label, axis=-1))[:2]] =
value
        label_mask = label_mask.astype(int)
    return label_mask

```

Andando a stampare i valori univoci della matrice, troviamo le classi univoche presenti all'interno della nostra immagine.

```

print(np.unique(mask))
[ 0  7  8  9 13 15 16 17 19 20 21 22 28]

```

La funzione di decoding sarà utile nel momento in cui sarà necessario mostrare l'immagine in output dalla rete neurale.

## DECODING

```

def decode_segmap(self,temp,n_classes,dictLabelRange):
    r = temp.copy(D)
    g = temp.copy()
    b = temp.copy()
    for l in range(0, n_classes):
        r[temp == l] = dictLabelRange[l][0]
        g[temp == l] = dictLabelRange[l][1]
        b[temp == l] = dictLabelRange[l][2]

    rgb = np.zeros((temp.shape[0], temp.shape[1], 3))
    rgb[:, :, 0] = r / 255.0
    rgb[:, :, 1] = g / 255.0
    rgb[:, :, 2] = b / 255.0
    return rgb

```

Una volta raccolta le classi nel dataset

```

train_data = ImagesDataset(
    path_data,
    'train',

```

```
customSize,  
fullLabelColor  
)
```

Definiamo le epoche e il batch size con cui la rete comincerà la fase di training.

Stabiliamo anche un learning rate iniziale, che, epoca dopo epoca, verrà raffinato con un decay impostato.

Nel file *metrics\_eval.py*, troviamo la funzione di train.

La funzione di ploss utilizzata per la segmentazione delle immagini è la CrossEntropyLoss (pixel-wise). Questa esamina ogni pixel individualmente, confrontando le istanze classe predette con il nostro vettore target. Questa loss è molto usata al giorno d'oggi, ma non considera un peso per ogni tipo di istanza. Questo può portare problemi nel caso di dataset sbilanciati

Un'altra funzione di loss popolare per la SSI delle immagini si basa sul coefficiente di Dice, che è essenzialmente una misura di sovrapposizione tra due immagini. Questa misura varia da 0 a 1, dove un coefficiente di Dice pari a 1 indica una sovrapposizione perfetta e completa.

Questa misura verrà usata come strumento di valutazione a posteriori.

Affiancata a questa loss di partenza, useremo un metodo di “modifica” della loss a seconda di alcune condizioni. Queste nel caso specifico sono le epoche di training.

```
scheduler_lr = StepLR(optimizer, step_size=2, gamma=0.1)
```

Con step size indichiamo ogni quante epoche attuare la modifica e con 0.1 di quanto moltiplicare il valore utilizzato in precedenza

```
x=batch['image'].to(device)  
y=batch['label'].to(device)  
output = model(x)  
n = x.shape[0] #numero di elementi nel batch  
global_step += n  
l = criterion(output,y)
```

```
if mode=='train':  
    l.backward()  
    optimizer.step()  
    optimizer.zero_grad()
```

Al fine di ogni epoca attuiamo il nostro lr\_decay.

Durante la fase di training è possibile consultare tensorboard per tracciare l'andamento della loss nel corso del tempo.

Ogni rete prende in input due parametri principali affinché sia possibile addestrare la rete sulle giuste immagini. Questi sono i canali d'ingresso (RGB nel caso) e le classi uscenti (intesi come channel nel modello d'uscita)

In generale, questo è specificato nel costruttore della generica classe.

```
class R2U_Net(nn.Module):  
    def __init__(self, img_ch=3, output_ch=19):
```

## Valutazioni e metriche

Le valutazioni sono state effettuate sul dataset di test/validation, a seconda della grandezza degli esempi da analizzare.

Nel nostro caso si è scelto di usare il validation set

```
val_data = ImagesDataset(  
    path_data,  
    'val',  
    customSize,  
    fullLabelColor  
)
```



Per quanto riguarda il numero di classi da predire / identificare, si è attuato un accorpamento di alcune label, soprattutto quelle relative agli unknown/void.

Questo ha portato un numero totale di classi pari a 31.

Le metriche con cui andremo a comparare le nostre CNN sono ben 4: la maggior parte di queste si basa sulla confusion matrix e sul calcolo di True Positive, False Positive etc..

Tra queste troviamo

1. Pixel wise accuracy
2. Jaccard index
3. F1 score
4. Dice score

Andiamo a vedere cosa trattano queste e quali sono i punti di forza/debolezza per una corretta comprensione.

Per semplificare gli esempi e la spiegazione faremo in modo che il problema trattato sia binario, e quindi due classi da rilevare: cane o gatto, oppure sfondo o persona

La "pixel-wise accuracy" (accuratezza pixel per pixel) è una metrica comune utilizzata per valutare le prestazioni di reti CNN, inerenti alle immagini.

Questa calcola la percentuale di pixel correttamente classificati rispetto al numero totale di pixel nell'immagine. Formalmente, si calcola come:

$$\text{Pixel Accuracy} = \frac{\text{Number of correctly classified pixels}}{\text{Total number of pixels}}$$

*Figura 12 Formula accuracy*

Tuttavia, per quanto possa essere la metrica più immediata e facile da calcolare soffre di un problema di bilanciamento delle classi. È facile intuire come, più un dataset è sbilanciato verso una particolare categoria, più sarà facile identificarla nel totale delle classi. Vedi ad esempio la strada nel dataset cityscape.

Per evitare questo problema, ad accompagnare la accuracy è la F1 score. Essa combina altre due metriche ottenute dalla Confusion Matrix, ovvero il recall e la precision. In generale, più è alta la F1 score più si avrà una buona predizione.

$$F1\ Score = 2 \times \frac{recall \times precision}{recall + precision}$$

Figura 13 Formula f1 score

Le Altre metriche comunemente utilizzate insieme alla accuracy includono l'Intersection over Union (IoU) e la Dice Coefficient.

L'IoU misura la percentuale di intersezione tra la maschera predetta e la maschera di ground truth rispetto all'unione delle due, mentre il coefficiente di Dice calcola il rapporto tra il doppio dell'area di intersezione e la somma delle aree della maschera predetta e di quella di ground truth.

Spiegando quanto sopra in parole semplici, misuriamo quanto la maschera e la sottoparte dell'immagine si intersichino tra di loro.

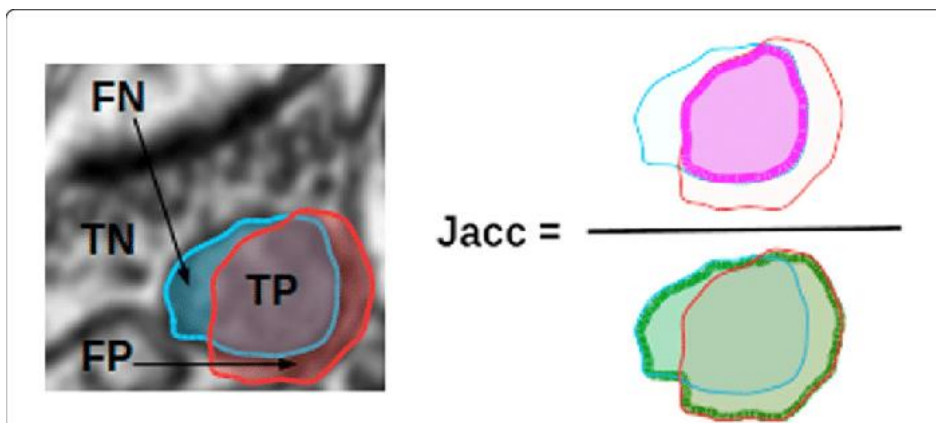


Figura 14 Formula jaccard index

Il dice score è molto simile al jaccard index, essa infatti manipola in maniera diversa unione e intersezione. Viene più indicato come indice di similarità tra due regioni piuttosto che di intersezione e spesso viene preferito a jaccard. Da questa particolarità, infatti, è nata anche la DICE LOSS come funzione di loss durante la fase di training.

$$DSC = \frac{2 |X \cap Y|}{|X| + |Y|}$$

Figura 15 Formula dsc score

Per il seguente task verrà utilizzata la DICE score, in quanto, nel corso del tempo si è verificata essere la più affidabile per il task di image segmentation.

Prima di calcolare matematicamente le metriche da noi scelte è necessario trasformare i tensori in numpy array.

Per fare questo ci serviamo, in particolar modo della funzione max di pytorch. Ricordiamo che il nostro output è suddiviso in 31 canali, ed è necessario prendere il massimo dai canali uscenti. Massimo che coincide con l'intero predetto dalla rete (entità semantica). È molto importante che i due vettori abbiamo la stessa dimensione

```
pred = output.data.max(1)[1].cpu().numpy()
gt = y.data.cpu().numpy()
```

Ogni metrica calcolata viene aggiornata per ogni iterazione (batch size) dentro un'epoca. Di questa viene fatto un valore medio e infine viene calcolata la media delle medie, come unica misura.

Per quanto riguarda l'indice di accuracy è stato scelto di affidarsi a sklearn.metrics

```
acc = skm.accuracy_score(true_label, predicted_label, normalize=True)
```

Invece, per calcolare il Dice score è possibile usare la variante di calcolo con formula booleana

$$DSC = \frac{2TP}{2TP + FP + FN}$$

Figura 16 Formula booleana dsc

Per fare questo dobbiamo collezionare e aggiornare una confusion matrix all'interno di un'epoca. Ci serviamo della classe "RunningScore", che permette di costruire e aggiornare un istogramma man mano che passiamo i due vettori.

Per poter poi ottenere le varie sotto misure (TP,FP,FN,TN) utilizziamo l'istogramma calcolato.

```
hist = self.confusion_matrix

TP = np.diag(hist)
TN = hist.sum() - hist.sum(axis = 1) - hist.sum(axis = 0) + np.diag(hist)
FP = hist.sum(axis = 1) - TP
FN = hist.sum(axis = 0) - TP
```

E da qui estraiamo due score, la f1 e il dice

```
# F1 = 2 * Precision * Recall / Precision + Recall
f1 = (2 * prec * sensti) / (prec + sensti + 1e-6)

#Dice formula
dsc_cls = (2* TP) / ( (2* TP) + FP + FN + 1e-6)
```

Per evitare divisione per 0, nel peggiore dei casi, è stato aggiunto una piccola quantità al denominatore

Tutte le reti utilizzeranno le seguenti metriche come metodo di comparazione.

## Benchmarking

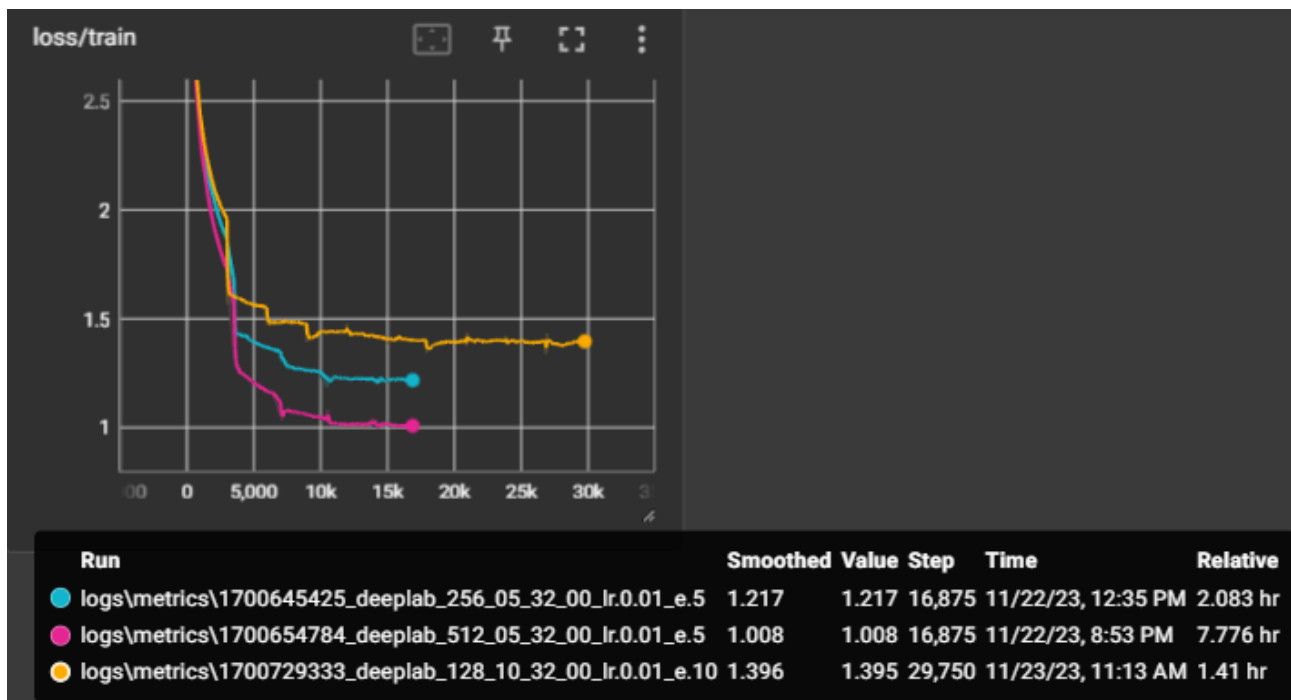
In questa sezione vedremo a confronto le varie CNN.

La loss iniziale per ogni benchmark è stata dello 0.01, con uno step size di 2 epoche e con un gamma di 0.1 (di quanto effettuare la divisione) ogni step size.

```
scheduler_lr = StepLR(optimizer, step_size=2, gamma=0.1)
```

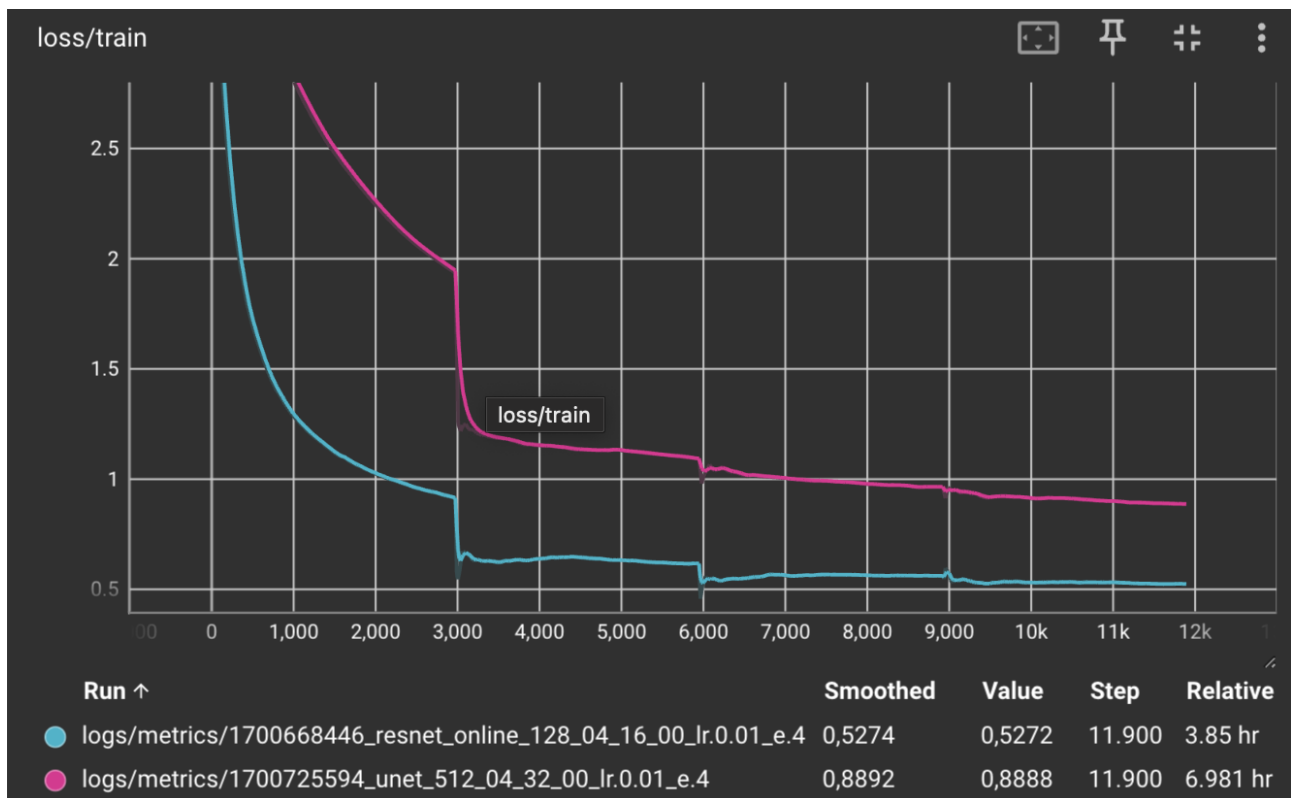
Le label dei logs delle matriche sono così formattate:

```
nomerete_sizeImg_epoch_lr_e.(epochs) //refuso
```



Andando ad esaminare prima la loss del deeplab con diverse configurazioni, notiamo subito come vi è un decremento maggiore e più convincente nella configurazione più “onerosa”, ovvero quella che presenta un’immagine di size 512x512 e 10 epoche di training.

Tuttavia, nel generale andamento delle loss, l’aumento del numero di sample di training non sembra aiutare la rete nella convergenza verso un risultato ottimale. Di fatti, la loss rimane sopra il valore di 1.

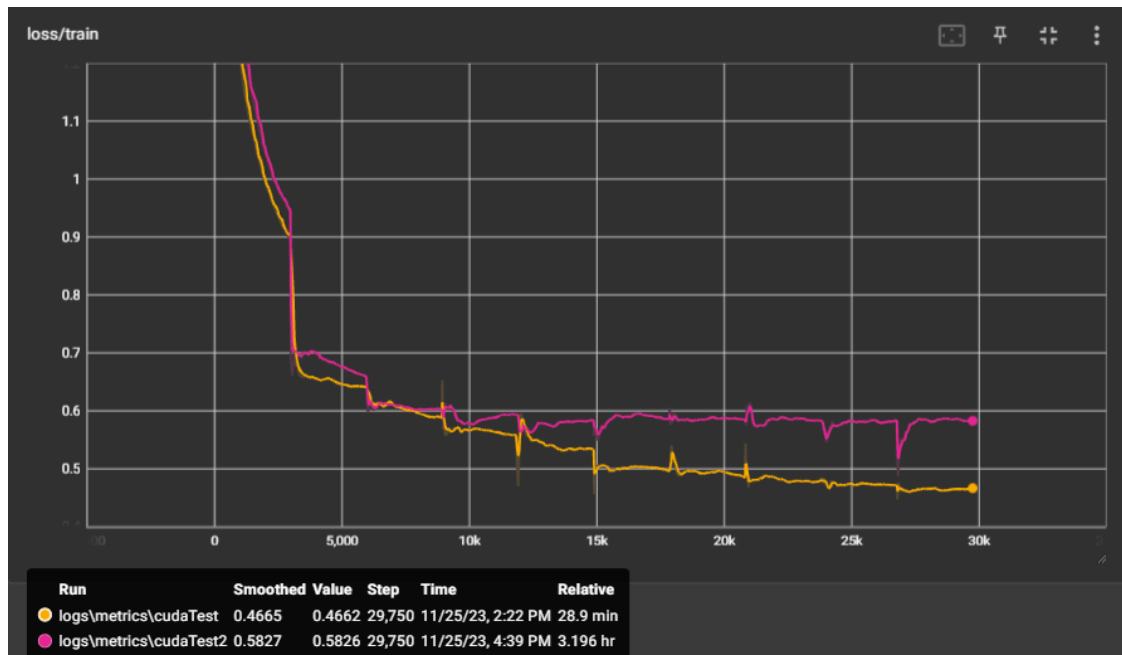


In quest'altro comparing è possibile notare come vi è una rapida convergenza della loss in entrambe le CNN (unet e variante resnet).

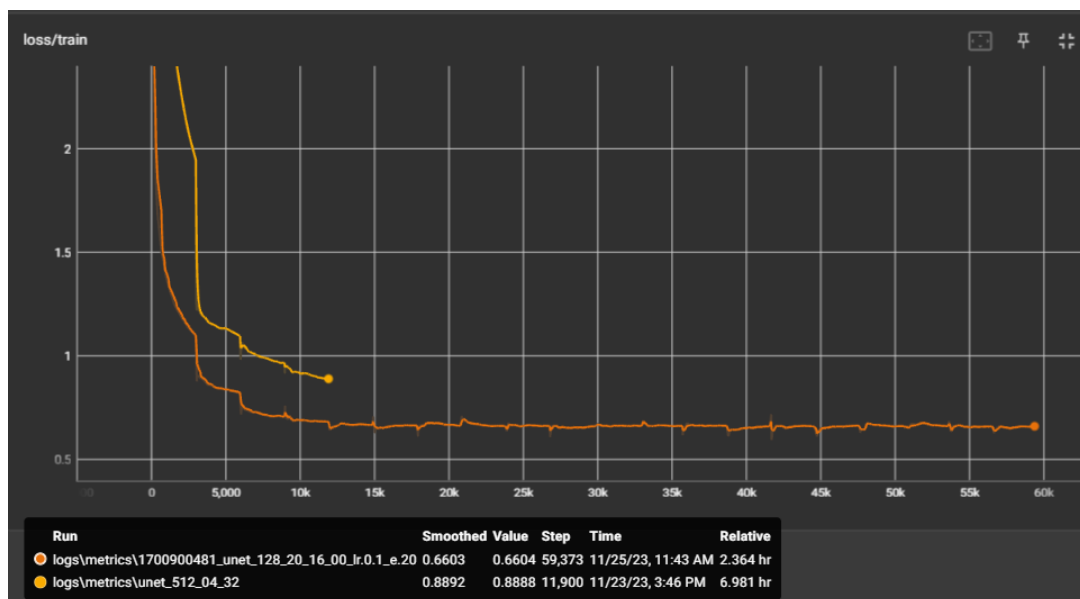
La resnet, nonostante abbia fatto un training con una size minore e con un numero di batch inferiore a quello della unet, arriva ad un valore di loss minore della sua rivale, segno che l'implementazione residual apporta i suoi benefici.

Si è provato inoltre a provare la rete unet con una macchina dotata di cuda, in particolare una NVIDIA 3070 TI 8 gb.

L'esecuzione ha portato il minor tempo di addestramento, solo ben 28 minuti rispetto ad una cnn allenata su CPU.



Provando ad inserire un numero di epoche maggiore nella rete UNET, rispetto alle precedenti si nota come la loss ondeggi nel tempo, non è un brutto segnale rispetto al totale appiattimento, segno che evidentemente serve un maggior tempo per allenare la rete.



In questo esempio si ottiene la dice più alta, ovvero 0.13.

Le metriche calcolate sotto sono calcolate come media delle medie, dove, a fine di ogni singola epoca trascorsa (in cui vengono visti X batch size), viene generato un numero parziale.

Al termine di tutte le epoche, per ogni tipologia di metrica, viene calcolato il numero finale.

CNN	ImgSize	Epoche	BatchSize	trainingTime(s)	ACC	F1S.	DiceS.
Deeplab	128	10	32	5083	0.73	0.11	0.04
Deeplab	512	5	32	28803	0.76	0.12	0.11
Unet	256	5	32	7210	0.76	0.1	0.1
Deeplab	256	5	32	7723	0.65	0.08	0.08
Unet	512	4	32	25198	0.77	0.11	0.1
ResUnet	128	5	32	13877	0.51	0.06	0.04
Unet	128	20	16	8554	0.81	0.13	0.13
ResUnet (cuda)	64	10	12	1738	0.66	0.11	0.09
ResUnet	64	10	16	11542	0.71	0.11	0.09

È possibile notare come la accuracy si rileva sempre alta, condizionata molto probabilmente dal bias della strada, la più facile da classificare, o dal cielo, molto grandi e statici rispetto al resto delle classi.

La f1 score e il dice score si rilevano essere invece molto bassi. Essendo i parametri più importanti per questo tipo di task, è bene fare affidamento a questi per calcolare la performance generale delle CNN.

La dice score sembra essere proporzionale a due parametri, il numero di epoche e anche la size dell'immagine. Il che torna con la logica della CNN e della procedura di training.

In generale tra le varie CNN, sembra che sia la deeplab che la unet base siano una buona base per partire, sia a livello di metriche che di tempistica.

L'implementazione resnet della unet, per quanto possa dare benefici, è sconsigliabile in quanto non sempre porta una buona metrica non un buon training time, tranne nel caso dell'unico case con la CUDA.



Nei due confronti con la size di 512 tra unet e deeplab le due reti sono abbastanza simili tra loro e hanno performance maggiori delle altre.

### Conclusione e considerazioni

Nel seguente progetto si è voluto analizzare, risolvere e comparare i risultati di alcuni approcci della computer vision odierna per il task della semantic segmentation.

Ogni rete è nata per risolvere un problema di dominio specifico, ma tutte possono essere adattate per risolvere task simili, come visto per la rete UNET, nata per il campo medico, ma ampiamente utilizzata per la sua semplicità. Le altre cnn implementano una loro particolarità, in grado di ampliare e migliorare le loro capacità, come la deeplab che adotta dei layer di convoluzione diversi dalle altre.

In generale, per poter apprezzare il meglio da ogni rete, sarebbe utile effettuare i relativi test su una macchina più capace di quella usata.

L'assenza di una GPU dotata di cuda ha giocato un ruolo fondamentale in questi test di benchmark.

In questo articolo: <https://www.dataenergy.ca/post/cpu-vs-gpu-computation-time-test-on-tabular-data>, è possibile vedere alcuni benchmark di confronto tra una rete allenata su GPU e una allenata su CPU.

Vi sono miglioramenti sotto ogni fronte, a partire da un training time ridotto, ad una gestione migliore della moltiplicazione di matrici, maggior utilizzo della banda, la possibilità di avere più epoche e maggiori batch size, molto importanti affinché la rete percepisca più dettagli possibili da immagini diverse.

Per quanto riguarda i risultati delle metriche, facendo affidamento maggiormente agli score di jaccard e dice, nessuna rete spicca nelle performance. Gli score si rivelano molto bassi.

Nei vari test condotti si è potuto notare come un crescente aumento delle epoche ha portato benefici in generale alle reti, tuttavia, a causa della ram ridotta, non si è potuto usare delle batch size maggiori. Infatti, la massima configurazione utilizzabile è stata quella della unet, la rete più semplice con una size di 512.

Il numero delle classi in ingresso è piuttosto alto e per poter captare le caratteristiche di ogni classe è necessario vedere un gran numero di esempi. Piuttosto, potremmo semplificare il problema escludendo gran parte delle classi,

come muri, alberi, cielo, grattacieli e lasciare quelle maggiormente rilevanti per un task di SSAI, come auto, umani e semafori. Inoltre, potremmo raggruppare alcune classi simili e difficili da distinguere in reti banali, come ad esempio le labels “drivers” e “humans”.

L'operazione di resize, necessaria per poter trainare la rete, è un'operazione molto comune durante la fase di preprocessing di immagini. Le immagini originali sono appartenenti al mondo reale, e per una corretta applicazione e riproducibilità dei risultati sarebbe bene lasciare la size invariata.

Questo è conveniente anche per rispettare i contorni delle entità semantiche, un essere umano, rispetto ad una strada, ha meno probabilità di essere associato con la giusta regione. Infatti, con una operazione di resize accade una perdita di informazioni spaziale e un cambiamento nel rapporto d'aspetto classico di una entità. Esistono degli approcci relativi anche a immagini multiscala, utili quando, è davvero necessario dover fare resize dell'immagine, spesso dovuto alle risorse limitate su cui gira la rete.

Su kaggle è possibile confrontare i test ottenuti, usando il medesimo dataset seguente link con il resto della community:

<https://www.kaggle.com/dansbecker/cityscapes-image-pairs/code?datasetId=22655>

Per quanto riguarda l'approccio al task di image segmentazione le reti CNN non sono l'unico approccio esistente.

Per dare un esempio, la seguente rete: <https://sampl-weizmann.github.io/DeepCut/>, utilizza una logica basata sui grafi. Questa rientra nelle generiche reti basate sulla GNN, in cui le relazioni tra pixel o regioni sono rappresentate come un grafo.

Un altro nuovo approccio molto utilizzato negli ultimi anni è l'uso dei Vision Transformer VT.

## Usage Over Time

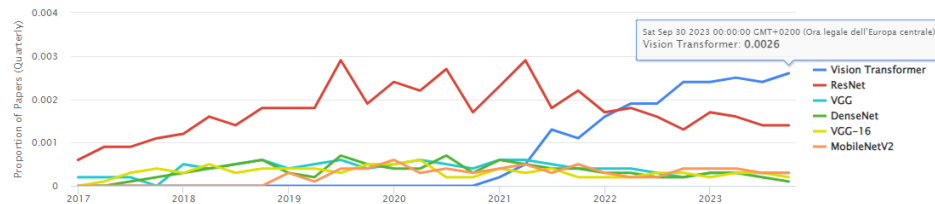


Figura 17 Model Usage Over year

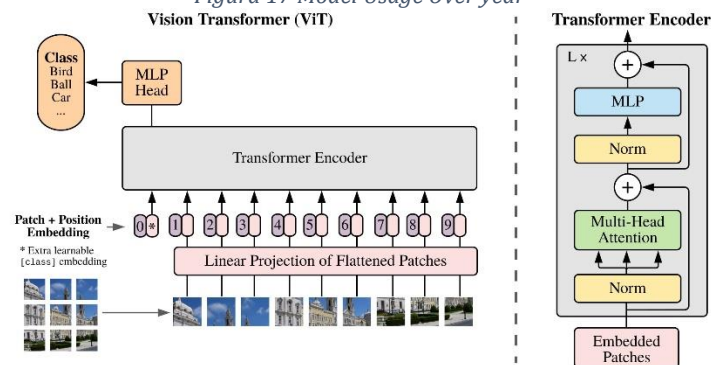


Figura 18 Transformer Visual

Un'immagine viene suddivisa in patch di dimensione fissa, ciascuno di essi viene quindi incorporato dentro un vettore di patch, viene aggiunta anche la loro posizione e inviata ad un Transformer.

Aldo Fiorito