Università degli studi di Modena
e Reggio Emilia

Dipartimento di Scienze e Metodi dell'Ingegneria

Corso di Laurea in Ingegneria Meccatronica

Creazione e training di una intelligenza artificiale per segmentazione semantica della rete stradale

Relatore:
Prof. Lorenzo Sabattini

Tesi di Laurea di:
Andrea Alboni

Correlatori:
Dott. Ing. Mattia Catellani

Anno Accademico 2022/2023

"L'uomo deve perseverare nell'idea che l'incomprensibile sia comprensibile; altrimenti rinuncerebbe a cercare."

J. W. Goethe

Sommario

Lo studio presentato in questa tesi è stato effettuato per rispondere alla necessità di selezionare una zona di interesse, quale per esempio la rete stradale, da un'immagine scelta dall'utilizzatore da integrare con algoritmi di controllo di agenti per ottenerne il coverage mediante l'impiego di un sistema multi-agente. Questa tesi si pone, dunque, come obiettivo la creazione e il training di un intelligenza artificiale capace di svolgere un task di segmentazione semantica binaria. In particolare, l'algoritmo viene addestrato affinché sia in grado di riconoscere la rete stradale presente in un'immagine. Tale predizione viene successivamente utilizzata per definire una funzione di densità di probabilità che interfacciata con un algoritmo di coverage consente di ottenere, tramite il controllo degli agenti di un sistema multi-robot, la copertura ottimale del sistema viario. Nello specifico, il lavoro presentato verte sul training e sull'implementazione della rete neurale come sullo sviluppo della distribuzione gaussiana e sulle simulazioni effettuate. U-Net è l'architettura scelta della rete neurale date le sue ottime prestazioni nello svolgimento di task simili. Per approssimare al meglio la regione di interesse è stato calcolato un Gaussian Mixture Model mentre per il controllo degli agenti è stato implementato un algoritmo di coverage centralized. Le simulazioni mostrano che la rete neurale sviluppata è adeguata per il compito e integrata con un algoritmo per il controllo di agenti permette di fornire una soluzione al problema analizzato.

Indice

Capitolo 1

Introduzione

Questo primo capitolo presenterà il problema analizzato, gli obiettivi dello studio e saranno fornite le nozioni teoriche di base per la comprensione della trattazione e dei risultati.

1.1 Introduzione al problema

La segmentazione di un'immagine, nell'ambito della visione artificiale, è l'operazione di partizione di un'immagine in regioni significative. Più precisamente, è il processo con il quale si classificano i pixel dell'immagine che hanno proprietà o caratteristiche comuni, ad esempio: colore, intensità o texture, e che pertanto appartengono alla stessa regione. Il risultato di un'immagine segmentata è, dunque, un insieme di regioni che, collettivamente, coprono l'intera immagine.
La segmentazione semantica è utilizzata per localizzare oggetti e bordi o come strumento per l'analisi delle immagini. Lo scopo è semplificare e cambiare la rappresentazione delle immagini in qualcosa che è più significativo e facile da analizzare. In questo contesto si colloca lo studio effettuato che si pone come obiettivo la realizzazione di una intelligenza artificiale in grado di estrarre, da una foto scelta dall'utente, la rete stradale al fine di disporvi in maniera ottimale degli agenti, droni.
Per la rete neurale si è optato per U-Net, una rete neurale di tipo convoluzionale, creata inizialmente per la segmentazione di immagini biomediche e ora diffusa per qualsiasi applicazione di segmentazione semantica, date le sue eccellenti

prestazioni.

1.2 Nozioni fondamentali

Le reti neurali convoluzionali, CNN o ConvNet, sono una classe di reti neurali specializzate nell'elaborazione di dati con una topologia simile a una griglia, ad esempio un'immagine.
Nelle CNN vengono applicati dei filtri il cui obiettivo è riconoscere particolari correlazioni o schemi ricorrenti all'interno dell'immagine stessa, al fine di generare delle caratteristiche ottimali da fornire poi in input alla rete neurale.
Sono composte da tre tipi principali di layer, che sono: il layer convoluzionale, il layer di pooling e il layer interamente connesso.

- **Layer Convoluzionale**: Dal punto di vista matematico, l'operazione di convoluzione, o correlazione incrociata, consiste nel moltiplicare tra loro due matrici, una rappresenta l'immagine oggetto di analisi mentre l'altra il filtro che viene applicato o Kernel, opportunamente traslate per calcolare la risultante ovvero la feature map, come mostrato in Figura 1.1. Un layer convoluzionale è definito da due parametri: il padding, p, ovvero il numero di zeri aggiunti intorno all'input, l'immagine, che ne aumentano le dimensioni e lo stride, s, la quantità di cui il kernel viene traslato quando scorre sull'immagine di input. Per una data dimensione dell'input (i), kernel (k), padding (p) e stride (s), la dimensione della feature map di output (o) generata è data dall'espressione 1.1:

$$o = \frac{i + 2p - k}{s} + 1 \tag{1.1}$$

  Al termine di questo processo si ottiene la feature map, una matrice di dimensione minore rispetto all'immagine di input.
  Nel processo di convoluzione si perdono alcune informazioni sull'immagine ossia quelle non contenute nel kernel prescelto, tuttavia, utilizzando più filtri di convoluzione si otterranno molteplici feature maps, ciascuna delle quali memorizzerà caratteristiche distintive dell'immagine in ingresso. É importante sottolineare che durante l'addestramento della rete neurale, questi kernel, i cui

elementi sono detti weights, vengono appresi tramite un metodo di ottimizzazione.

L'output del layer convoluzionale è la somma di ciascuna delle feature maps con il rispettivo bias, valore aggiornato durante il training della rete neurale utilizzato per compensare il risultato.

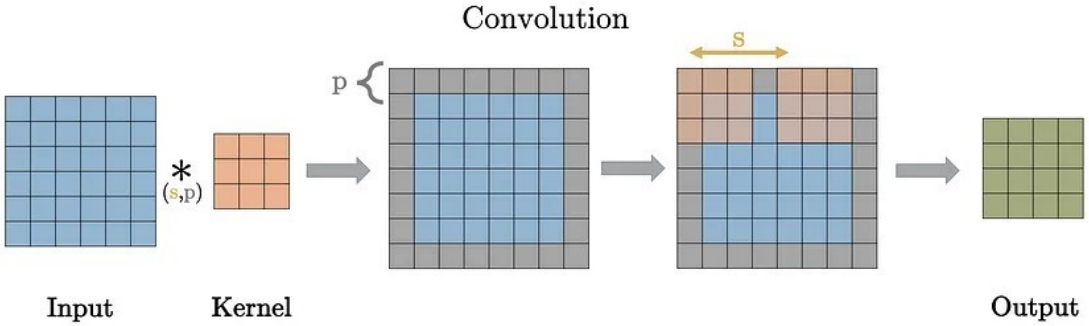

Figura 1.1: Esempio dell'operazione di convoluzione

- **Layer di Pooling**: L'output del layer convoluzionale è sotto-campionato attraverso un processo che prende il nome di pooling. Ciò comporta una riduzione dimensionale che consente di diminuire la complessità di una CNN considerando solo una parte dei dati.

  Due sono le tipologie di pooling, mostrate in Figura 1.2: max pooling e average pooling, ovvero si effettuano operazioni di calcolo del massimo o del valore medio su un subset delle entrate della matrice in input.

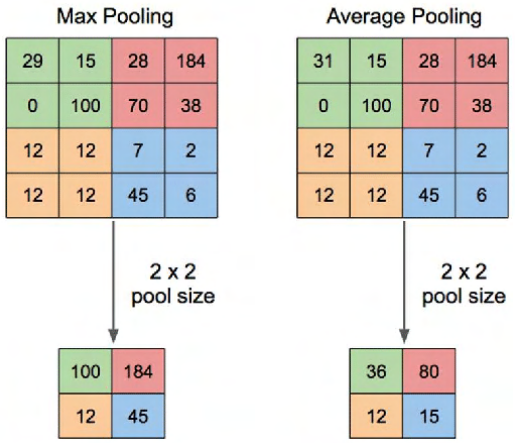

Figura 1.2: Esempio di Max e Average Pooling

- **Layer interamente connesso**: Il layer interamente connesso riceve in input l'output dei layer precedenti, ovvero le pooled feature map, lo appiattisce,

processo di flattening, trasformandolo in un vettore monodimensionale che sarà l'input per la fase successiva ovvero la ricostruzione dell'immagine.

Per classificare a livello di pixel è necessario però aggiungere un'implementazione inversa della CNN ovvero occorre addizionare dei layers che effettuino il sovra-campionamento, dei transposed convolutional layers. Ciascuno di questi è definito da due parametri: il padding, p, e lo stride, s, e permette di generare una feature map di output che ha una dimensione maggiore di quella in input. Il loro funzionamento, mostrato in Figura 1.3, può essere scomposto in quattro step:

- **Step 1**: calcolo di due nuovi parametri z = s - 1 e p' = k - p - 1.
- **Step 2**: inserimento tra ogni riga e colonna della feature map in input di un numero z di zeri, ciò ne aumenta la dimensione.
- **Step 3**: applicazione del padding, come nel convolutional layer
- **Step 4**: esecuzione della operazione convoluzionale sulla matrice generata dallo step 3 con uno stride pari a 1

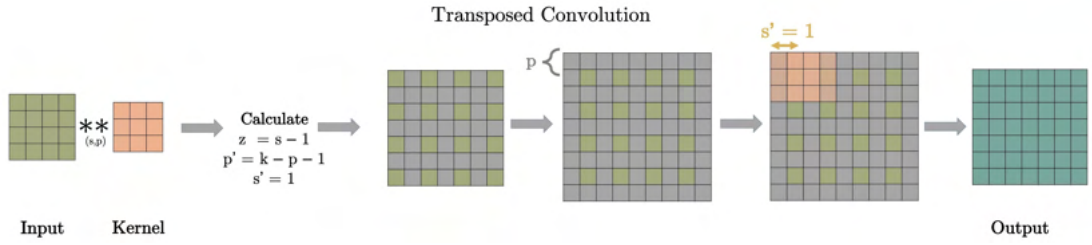

Figura 1.3: Transposed Convolutional Layer

L'idea alla base della transposed convolution è quella di eseguire un upsampling addestrabile, infatti come per l'operazione di convoluzione gli elementi del kernel sono aggiornati durante il training.
I processi di sovra-campionamento e sotto-campionamento vengono eseguiti lo stesso numero di volte per garantire che l'immagine finale abbia le stesse dimensioni dell'immagine di input. In questo modo si forma un'architettura di codifica/decodifica, come mostrato in Figura 1.4, che consente la segmentazione semantica.

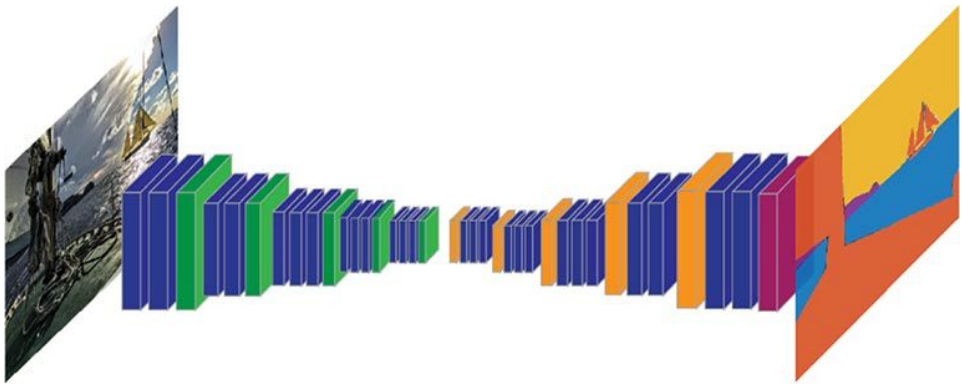

Figura 1.4: Architettura di una CNN per segmentazione semantica, a partire dalla codifica, sinistra, fino alla decodifica, destra, e la ricostruzione dell'immagine

Infine si giungerà al livello di output, dove sarà applicata una di due tipologie di funzioni di attivazione per completare la segmentazione dell'immagine: la funzione sigmoidea in caso di segmentazione binaria, oppure la sua variante multidimensionale chiamata softmax, adatta in caso di segmentazione relativa a molteplici categorie.

1.3 Ambiente di sviluppo

La CNN è stata sviluppata interamente in linguaggio python, nell'ambiente di sviluppo integrato Visual Studio Code. Ciò è avvenuto sfruttando TensorFlow, una libreria open source per l'apprendimento automatico e l'intelligenza artificiale. Uno dei motivi che ha determinato la scelta di TensorFlow è l'astrazione di alto livello consentita, che facilita la creazione, l'implementazione, l'addestramento e il test dei modelli di reti neurali complesse.

Capitolo 2

U-Net per estrazione della rete stradale

Le seguenti sezioni tratteranno della creazione della rete neurale, del tuning degli iperparametri e del training dell'intelligenza artificiale. Inoltre sarà affrontata la preparazione del dataset e le tecniche applicate per aumentare la portabilità della rete e garantire predizioni accurate.

2.1 Architettura U-Net

L'architettura U-Net, presentata in [1], contiene tre paths. Il primo è di down-sampling, chiamato anche di encoder, che viene utilizzato per estrarre le features dell'immagine. Il secondo path è quello di upsampling, chiamato anche di decoder, che ricostruisce la previsione partendo dall'output della fase di encoding. Il terzo percorso sono le skip connections, punto di forza di questa architettura. Come mostra la Figura 2.1, il modello ha la forma della lettera "U" da cui eredita il nome.

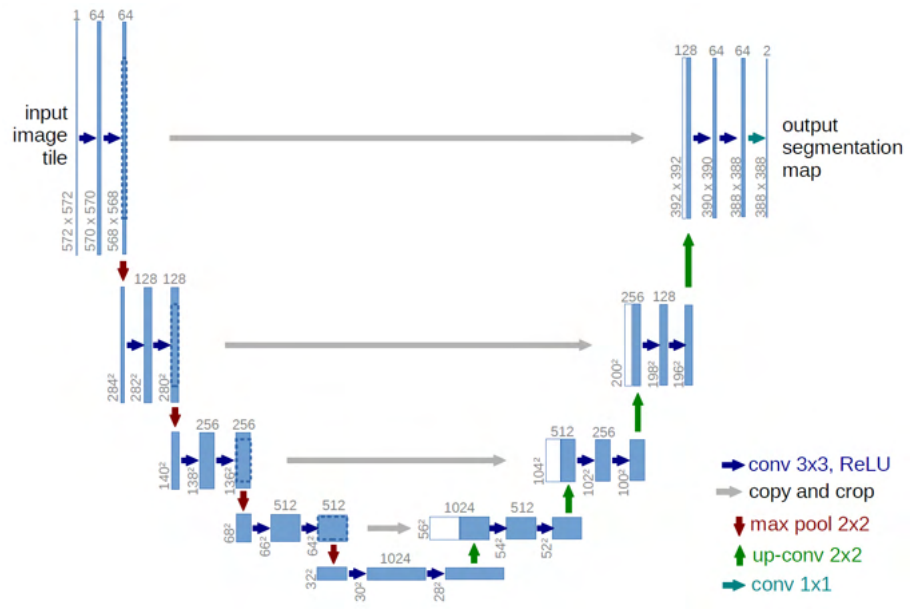

Figura 2.1: Architettura U-Net

Segue il codice 2.1 per la creazione dell'architettura U-Net organizzato in blocchi sviluppati in codice python sfruttando la libreria *keras.layers*. Il primo di questi implementa l'operazione di convoluzione, il secondo, composto dalla convoluzione seguita dal Maxpooling, è il path di encoder dell'architettura mentre il terzo è il path di decoder. In quest'ultimo si può notare la presenza delle skip connections realizzate dal comando *concatenate* di python.

```
def conv_block(input, num_filters):
    #Creazione blocco di convoluzione
    x = Conv2D(num_filters, 3, padding="same")(input)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = Conv2D(num_filters, 3, padding="same")(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    return x

def encoder_block(input, num_filters):
    #Creazione blocco di encoder
    x = conv_block(input, num_filters)
    p = MaxPooling2D((2, 2))(x)
    return x, p
```

```
def decoder_block(input, skip_features, num_filters):
    #Creazione blocco di dencoder
    x = Conv2DTranspose(num_filters, (3, 3), strides=2, padding="
        same")(input)
    x = Concatenate()([x, skip_features])
    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = conv_block(x, num_filters)
    return x
```

Come accennato in precedenza, i pooling layers utilizzano un metodo predefinito per creare una rappresentazione compatta dell'immagine di input, un esempio è il max pooling. A differenza, gli upsampling o deconvolution layers aumentano la dimensione utilizzando una funzione che viene aggiornata durante il training del modello. Nelle altre architetture, l'encoder è in grado di trasmettere le caratteristiche al decoder, ma la posizione di tali features, ovvero da quale livello di encoding provengono, è persa. Per risolvere questa mancanza, è necessaria un'enorme quantità di dati. Per ovviare a questo problema, l'architettura U-Net presenta invece delle skip connections. Queste consentono di trasmettere le features apprese durante la contrazione dell'input e la loro posizione al decoder per ricostruire l'immagine segmentata. Ciò preserva l'integrità strutturale dell'immagine aumentando l'accuratezza della previsione con un numero di dati necessari per il training notevolmente inferiore.

2.2 Dataset e training

Un dataset è definito come un insieme strutturato di dati organizzati in modo tale da poter essere facilmente elaborati da un algoritmo. Nel caso della segmentazione semantica, il dataset è composto da immagini, nel caso analizzato foto aeree, e le relative maschere ovvero immagini in cui i pixel potranno essere bianchi, se appartenenti a strade, o neri, se non appartenenti a strade. Un esempio di immagine e maschera associata è presentato in Figura 2.2.

Figura 2.2: Immagine a sinistra e la relativa maschera a destra

Il dataset utilizzato per il training della rete neurale proviene dalla DeepGlobe 2018 Satellite Image Understanding Challenge [2], è stato sviluppato per compiti di segmentazione, rilevamento e classificazione su immagini satellitari.

2.2.1 Preparazione del dataset

La prima operazione di preparazione del dataset è la binarizzazione delle maschere in quanto è richiesto che presentino solo pixels bianchi, 255, o neri, 0. Il set di dati scelto non verifica la condizione in quanto le tonalità dei pixels appartengono alla scala di grigi, con valori compresi tra 0 e 255. Una volta resi binari tali valori, si procede con una divisione in patches, quadrati di lato 256 ovvero della dimensione ottimale per segmentazione semantica mediante U-Net, sia delle immagini che delle maschere. La preparazione del dataset prevede inoltre la divisione delle immagini e delle masks in un sistema organizzato di cartelle, per rendere il più agevole la loro estrazione e lettura in fase di training. In particolare l'insieme delle immagini e delle relativi previsioni desiderate, è stato suddiviso in tre subdataset: training, validation e test dataset. La rete neurale durante il training riceverà come input da cui apprendere solo le immagini appartenenti al training dataset mentre quelle appartenenti al validation dataset saranno utilizzate per calcolare l'accuratezza e la capacità di generalizzazione della rete neurale. Diversamente quelle del test dataset saranno utilizzate, a training terminato, per testare il modello ottenuto.

2.2.2 Training

Il training della rete neurale consiste nella regolazione di una serie di parametri, corrispondenti ai differenti pesi da assegnare alle varie connessioni della rete e i rispettivi biases. Ciò avviene effettuando comparazioni della previsione, all'inizio del tutto arbitraria, con quanto suggerito dall'apprendimento, le masks. Dal confronto tra i valori calcolati dalla rete e i valori corretti, è stabilito il livello d'errore per ogni nodo che serve per correggere i pesi negli hidden layers della rete neurale. Iterando il processo sul dataset più e più volte, la rete impara ad aggiustare i pesi per produrre il corretto output. L'obiettivo del training è dunque quello di raggiungere un minimo della loss function, funzione di errore, mediante l'aggiornamento periodico dei parametri della rete. Il codice che svolge l'operazione di training della rete neurale è riportato a seguire.

```
def jaccard_coef(y_true, y_pred):
    #Calcolo del jaccard coefficient
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (intersection + 1.0) / (K.sum(y_true_f) + K.sum(y_pred_f
       ) - intersection + 1.0)

dice_loss = sm.losses.DiceLoss()
focal_loss = sm.losses.BinaryFocalLoss()
total_loss = dice_loss + focal_loss

model.compile(optimizer=Adam(0.00001, beta_1=0.99, beta_2=0.99),
   loss=[total_loss], metrics=[jaccard_coef])

callbacks = [tf.keras.callbacks.ModelCheckpoint('Model-{epoch}-{
   val_jaccard_coef}.h5', period=10), tf.keras.callbacks.
   EarlyStopping(monitor="val_loss", patience=10, start_from_epoch
   =0)]

history = model.fit(train_generator, validation_data=val_generator,
    batch_size=batch_size, validation_batch_size=batch_size,
   steps_per_epoch=steps_per_epoch, validation_steps=
   steps_per_epoch, epochs=250, callbacks= [callbacks] )
```

Come si può notare dal codice, la loss function utilizzata è la binary focal loss[1], addizionata alla dice loss, introdotta in [3]. Diversamente, l'accuracy durante il training è stata monitorata attraverso il coefficiente di Jaccard, un indice statistico utilizzato per confrontare la similarità e la diversità di insiemi campionari. Tale numero è definito come la dimensione dell'intersezione divisa per la dimensione dell'unione degli insieme che in questo contesto sono la predizione ottenuta e quella desiderata, la maschera. Stabilite le funzioni per monitorare il progresso della rete durante l'allenamento, si è applicata l'operazione di data augmentation alle immagini del training dataset prima che queste siano fornite come input alla rete neurale. Ciò consiste in una tecnica utilizzata per espandere artificialmente la dimensione del training dataset creando dati mediante la modifica di quelli esistenti. É una pratica diffusa poiché consente di aumentare la precisione delle reti neurali, che è fortemente legata alla quantità di dati a disposizione per l'addestramento, e poiché consente di evitare l'overfitting. Ovvero un comportamento negativo che si presenta quando la rete neurale analizzando il training dataset anziché imparare a generalizzare, lo memorizza. Un segnale di ciò è, in fase di training, l'aumento dell'accuratezza sul training dataset e la costanza di quella sul dataset di validazione. Le modifiche apportate alle immagini sono state scelte in funzione ai risultati di [4], sono il capovolgimento orizzontale e verticale, l'applicazione di uno zoom e il riempimento dell'immagine così ottenuta mediante specchiatura[2]. Tali modifiche verranno apportate anche alle rispettive maschere, ciò è garantito dall'utilizzo dello stesso *seed*. L'applicazione di tali alterazioni alle immagini e alle maschere è stata implementata nei seguenti passaggi.

```
seed = random.randint(0,100)

img_data_gen_args = dict(rescale = 1/255., zoom_range=0.3,
    horizontal_flip=True, vertical_flip=True, fill_mode='reflect')
```

[1]La focal loss si concentra sugli esempi che il modello sbaglia piuttosto che su quelli che può prevedere con sicurezza. Assicura che le previsioni su esempi difficili migliorino nel tempo piuttosto che diventare eccessivamente sicure con quelli facili attraverso la riduzione del peso, una tecnica che riduce l'influenza degli esempi facili sulla loss function con conseguente maggiore attenzione agli esempi difficili.

[2]In caso in cui il valore, scelto casualmente in un range, dello zoom rimpicciolisca l'immagine, per riportarla alla dimensione di input, 256 x 256, i pixels mancanti sono aggiunti specchiando l'immagine.

```
mask_data_gen_args = dict(rescale = 1/255., zoom_range=0.3,
    horizontal_flip=True, vertical_flip=True, fill_mode='reflect',
    preprocessing_function = lambda x: np.where(x>0.5, 1, 0).astype(
    x.dtype))
```

Il codice 2.2.2 mostra altre due operazione di preprocessing oltre a quelle appena discusse: la prima è la divisione del valore di ciascun pixel per 255 applicata sia alle immagini che alle maschere, effettuata siccome facilita la rete neurale durante il training, e la seconda è una correzione dei valori dei pixel delle maschere successivamente alle modifiche apportate dall'augmentation, applicata per garantire che tutti i pixel abbiano valore pari a 0 o 255. Un esempio delle immagini e maschere in input alla rete neurale è mostrato in Figura 2.3.

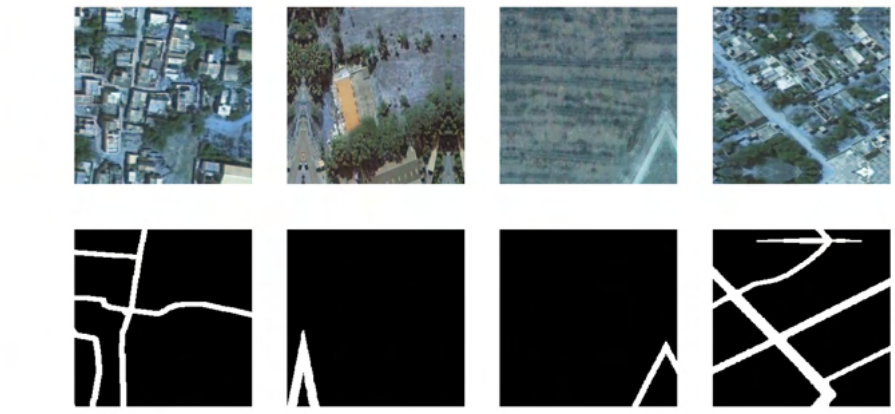

Figura 2.3: Data augmentation su immagini e maschere

Il training dataset così ottenuto, è di dimensioni troppo elevate per essere fornito come input alla rete, dunque viene suddiviso in batches. Ciò è stato implementato nelle righe di codice 2.2.2 seguenti le quali sfruttano la funzione *data_generator.flow_from_directory* per caricare da locale un batch di immagini e uno delle rispettive maschere.

```
batch_size = 16

image_data_generator = ImageDataGenerator(**img_data_gen_args)
```

```
image_generator = image_data_generator.flow_from_directory(
    dir_train_image_path, seed=seed, shuffle=True, batch_size=
    batch_size, class_mode=None)

mask_data_generator = ImageDataGenerator(**mask_data_gen_args)

mask_generator = mask_data_generator.flow_from_directory(
    dir_train_mask_path, seed=seed, shuffle=True, batch_size=
    batch_size, color_mode = 'grayscale', class_mode=None)
```

Ciascuno di questi batch è composto da immagini estratte in ordine casuale, *shuffle = True*, questa operazione è ripetuta fino a che non tutte le immagini appartenenti al sub-dataset di training sono state fornite come input alla rete neurale. Nel corso dell'addestramento, la rete neurale visionerà tutte le immagini destinate al training centinaia di volte. Pertanto, il completamento del training richiede diverse ore, a volte giorni, per questo sono state inserite delle callbacks, presenti nel codice 2.2.2. Le callbacks sono funzioni che si attivano durante il training e che consento di salvare il modello periodicamente o interrompere in anticipo il training in caso non si verifichi un miglioramento, determinato dal monitoraggio della *val_loss*, del modello dopo un numero definito, *patience*, dall'utente di iterazioni. Queste funzioni si sono dimostrate particolarmente utili per il confronto tra modelli di U-Net aventi un numero diverso di parametri e per il tuning degli iperparametri e sono state implementate nei seguenti passaggi.

2.2.3 Tuning degli iperparametri

Per la risoluzione di questo studio sono stati analizzati diversi modelli di U-Net in combinazione a diversi iperparametri o elementi capaci di influenzare il training come: la dimensione del batch, il learning rate, l'optimizer e il numero dei parametri.

- **Influenza del Learning Rate**: Il learning rate è un iperparametro che determina l'entità della modifica apportata ai parametri della rete neurale ad ogni iterazione durante il training, pertanto può avere un impatto significativo sulle prestazioni di un modello. Infatti, un elevato learning rate può comportare il superamento del minimo della loss function, come in Figura 2.4, e portare a

instabilità o oscillazioni durante il training. D'altra parte, un basso learning rate può comportare una lenta convergenza verso i parametri ottimi o portare a convergere a una soluzione non ottimale, minimo locale. Il learning rate può essere costante o regolato dinamicamente durante il training, a seconda dell'algoritmo di ottimizzazione utilizzato. Per il training della rete neurale sviluppata, è stato scelto l'optimizer Adaptive Moment Estimation (Adam), un algoritmo di ottimizzazione introdotto in [5]. Adam, è un algoritmo di adaptive learning rates ovvero calcola i learning rates individualmente per i diversi parametri del modello. Questo aiuta l'optimizer a convergere più velocemente e con maggiore precisione rispetto ai metodi classici a tasso di apprendimento fisso. Sono stati effettuati test con learning rate iniziale di valore: 0.001, 0.0001, 0.0002, 0.0005 e 0.00001.

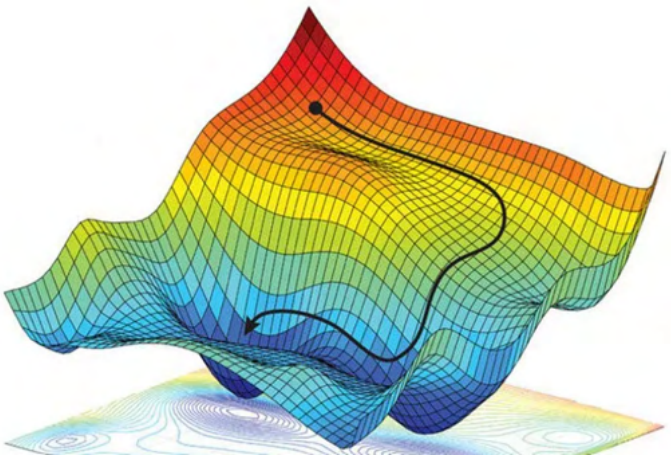

Figura 2.4: Esempio di una loss function e del percorso desiderato di apprendimento

- **Influenza del numero di parametri**: Il numero di parametri in un dato layer è il conteggio degli elementi capaci di apprendere per un filtro, in generale sono i pesi e i biases che vengono aggiornati durante l'allenamento mediante il processo di retro-propagazione. Il numero di parametri determina la complicatezza della rete neurale. Se la rete neurale è composta da pochi parametri rischia di non non essere in grado di apprendere dal dataset, mentre averne troppi può condurre velocemente all'overfitting. Sono stati testati modelli di U-Net con 2 milioni, 27 milioni, 31 milioni e 34 milioni di parametri.

- **Influenza del batch size**: L'analisi della dimensione del batch, ovvero il numero di immagini che sono fornite in input alla rete neurale contempora-

neamente, è condizionata dalle capacità di calcolo a disposizione. Sono stati effettuati test con batches di dimensioni di 16 e 32.

La scelta del numero di parametri e il tuning degli iperparametri sono stati svolti effettuando il training delle diverse combinazioni e tabulando il valore della validation accuracy raggiunta in funzione del numero di epoca[3]. In seguito a diverse prove si è stabilito che il valore ottimale del batch size fosse 16 ed è stato possibile determinare il miglior learning rate per ciascuna architettura. É riportata a seguire la tabella 2.1 che mostra l'accuracy percentuale di ciascuna architettura con il rispettivo miglior learning rate in funzione del numero di epoca ovvero la tabella che ha consentito di determinare la miglior combinazione architettura - iperparametri.

		Epoca							
Parametri	L.R.	1	5	10	15	20	30	40	50
2 Mln	0.0002	18.35%	46.83%	51.74%	52.94%	54.46%	55.01%	55.73%	56.93%
27 Mln	0.0001	13.97%	46.39%	51.15%	52.89%	54.03%	56.93%	57.82%	58.25%
31 Mln	0.0005	27.84%	36.83%	48.99%	52.10%	53.67%	54.20%	55.52%	55.72%
34 Mln	0.0001	15.18%	48.89%	53.97%	56.88%	57.24%	58.30%	57.87%	59.88%

Tabella 2.1: Confronto tra le architetture U-Net composte da un diverso numero di parametri e il rispettivo miglior learning rate al procedere del training

Dalla tabella si evince che i risultati migliori si possono ottenere continuando a trainare la U-Net composta da 34 milioni di parametri, con un batch size di 16 ad un learning rate di 0.0001. Proseguendo il training fino alle 100 epoche, si è raggiunta un'accuratezza sul set validazione del 61% con una loss sullo stesso del 30%. Continuare ulteriormente non porterebbe ad un miglioramento dei risultati a causa del fenomeno dell'overfitting. Tale rete neurale è stata ottenuta dalle seguenti righe di codice 2.2.3 le quali si basano sui blocchi di codifica e decodifica precedentemente presentati.

```
def build_unet(input_shape):
    #Creazione architettura U-Net
    inputs = Input(input_shape)

    s1, p1 = encoder_block(inputs, 32)
```

[3]Si definisce epoca l'intero passaggio del training dataset attraverso l'algoritmo

```
s2, p2 = encoder_block(p1, 64)
s3, p3 = encoder_block(p2, 128)
s4, p4 = encoder_block(p3, 256)
s5, p5 = encoder_block(p4, 512)

b1 = conv_block(p5, 1024)

d1 = decoder_block(b1, s5, 512)
d2 = decoder_block(d1, s4, 256)
d3 = decoder_block(d2, s3, 128)
d4 = decoder_block(d3, s2, 64)
d5 = decoder_block(d4, s1, 32)

outputs = Conv2DTranspose(1, 1, padding="same", activation="
    sigmoid")(d5)

model = Model(inputs, outputs, name="U-Net")
model.summary()
return model
```

2.3 Sviluppo delle predizioni

Come detto in precedenza, la dimensione dell'immagine richiesta in input dalla rete neurale è 256 x 256 pixels. Si è dunque effettuato uno studio per estendere la fruibilità della rete, garantendo una predizione accurata, anche a immagini con dimensioni maggiori.

- **Divisione in patches da 256 x 256**: L'immagine viene segmentata in sotto-immagini di lato 256 pixels che vengono fornite in input alla rete neurale. La predizione dell'immagine originaria viene quindi ricostruita combinando le predizioni ottenute sulle patches. Questo metodo si è dimostrato efficace in termini di accuratezza della previsione fino a che l'immagine di input aveva dimensioni inferiori di 1024 x 1024 pixels in quanto, per immagini più grandi, i patches risultavano porzioni di immagine poco significativi poiché si perdeva la contestualizzazione, come per esempio riportato in Figura 2.5.

- **Resize dell'immagine**: Il secondo metodo analizzato consiste nel ridimensionamento delle immagini in ingresso alla rete neurale. In particolare, entrambi

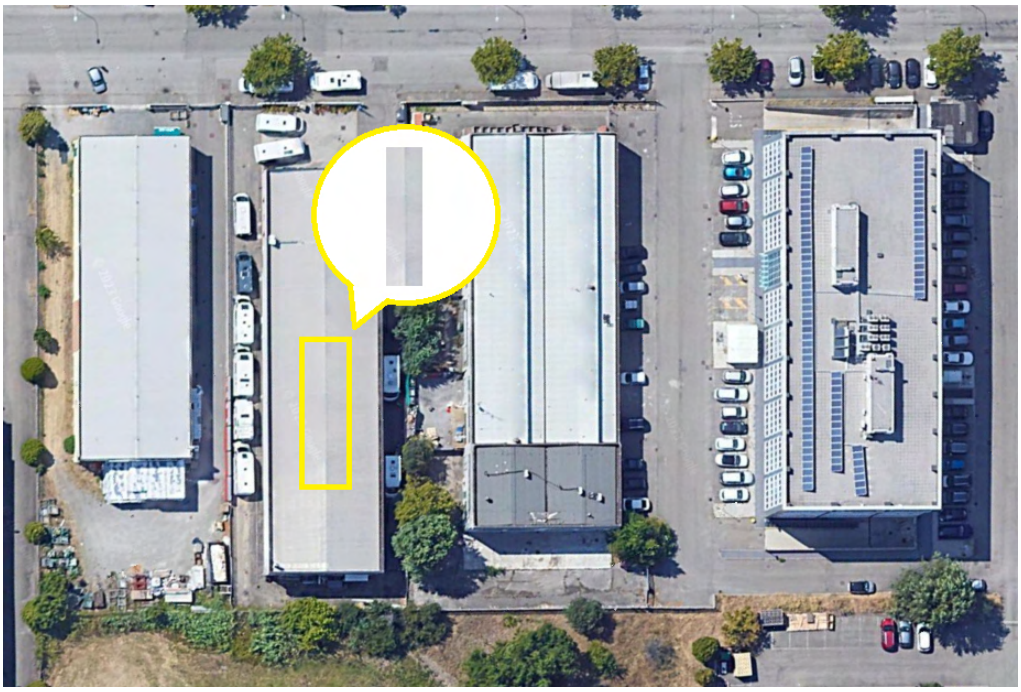

Figura 2.5: Esempio degli errori che la mancanza di contestualizzazione data dalla suddivisione in patches può causare

i lati vengono scalati dello stesso fattore determinato affinché il lato più corto dell'immagine abbia dimensione finale pari a 256 pixels. In caso in cui la dimensione del lato più lungo risulti maggiore di 256, allora viene applicato il metodo di divisione in patches presentato in precedenza. La predizione dell'immagine originaria viene quindi conseguita unendo le predizioni ottenute da eventuali patches e ridimensionando la predizioni utilizzando l'inverso del fattore identificato in precedenza. I risultati ottenuti risultano ottimi per immagini di grandi dimensioni, bensì, in quelle di dimensioni minori a 1024 x 1024 si notano alcune imprecisioni.

Dati i metodi appena trattati si è optato per applicarli entrambi: il primo per immagini di dimensioni minori a 1024 x 1024 e il secondo per immagini più grandi.
Risolto il problema della limitata portabilità della rete, con l'obiettivo di ottenere la miglior precisione possibile delle predizioni, si è optato per l'utilizzo di due strategie di previsione in sinergia per poi effettuare una media pesata dei risultati ottenuti.

- **Predizioni su patches specchiati**: Per ciascuno dei patch, di dimensioni

256 x 256, alla rete neurale viene fornito in input tale patch, tale patch specchiato orizzontalmente, verticalmente e rispetto all'origine, come in [6], un esempio di ciò in Figura 2.6. Le previsioni sono poi specchiate nella maniera opposta, processo che le riporta tutte nell'orientazione del patch originale, per generare la predizione finale ottenuta attraverso la media di queste. Attuando questa pratica, la rete neurale predice quattro volte su ciascun patch, visto in configurazioni diverse, questo consente di aumentare l'accuratezza delle previsioni, riducendo notevolmente il numero di falsi positivi, pixels che in una configurazione sembrano appartenere ad una strada.

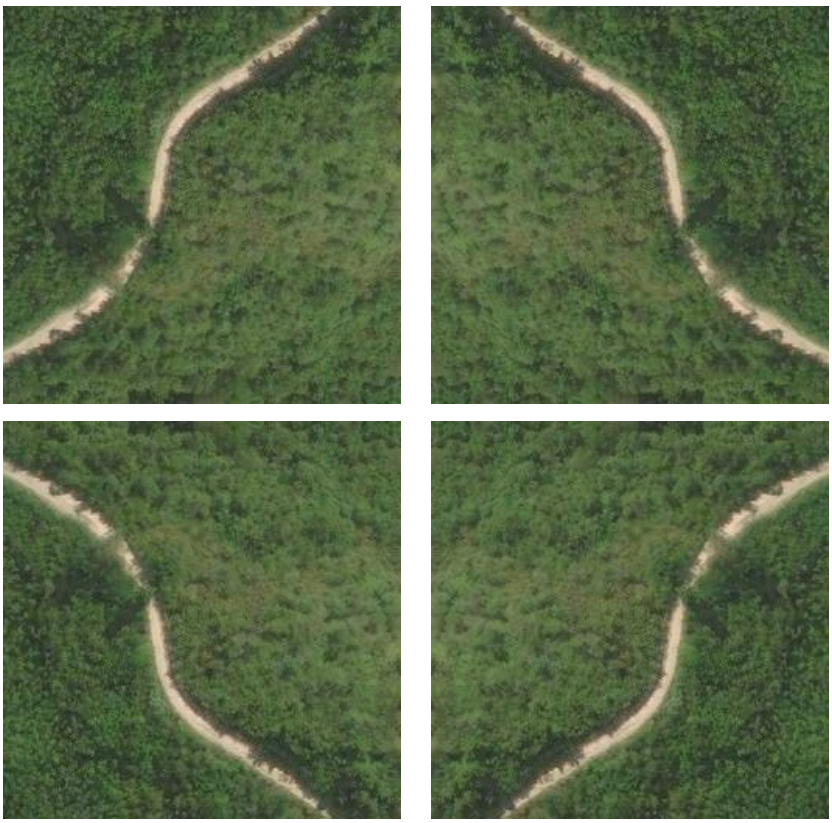

Figura 2.6: Patch originale in alto a sinistra, specchiatura orizzontale in alto a destra, specchiatura verticale in basso a sinistra e specchiatura rispetto all'origine in basso a destra

Questa strategia di predizione è stata implementata mediante i seguenti passaggi 2.3. *p0, p1, p2* e *p3* sono le quattro predizioni sviluppate sul patch originale e specchiato. Dalle ultime linee del codice riportato si nota il calcolo della media delle probabilità di appartenenza di ciascun pixel ad una strada, in caso questa media risultasse maggiore del valore di soglia, *tresh*, allora tale pixel sarà riconosciuto come appartenente ad una strada. Avere $tresh = 0.8$

significa che ciascun pixel di colore bianco nella predizione ha una probabilità di appartenenza ad una strada di almeno 80%.

```
mask_patches = []

for i in range(len(patches)):
    img = patches[i] / 255.0
    p0 = model.predict(np.expand_dims(img, axis=0))[0][:, :, 0]
    p1 = model.predict(np.expand_dims(np.fliplr(img), axis=0))
       [0][:, :, 0]
    p1 = np.fliplr(p1)
    p2 = model.predict(np.expand_dims(np.flipud(img), axis=0))
       [0][:, :, 0]
    p2 = np.flipud(p2)
    p3 = model.predict(np.expand_dims(np.fliplr(np.flipud(img))
       , axis=0))[0][:, :, 0]
    p3 = np.fliplr(np.flipud(p3))
    thresh = 0.8
    p = (p0 + p1 + p2 + p3) / 4
    mask_patches.append(p)

prediction = recompone_images(mask_patches)
#recompone_images : funzione per la ricostruzione dell'immagine
    dai patches
pred = (prediction > thresh).astype(np.uint8)
```

- **Blending patches**: La predizione viene effettuata su patches caratterizzati da un certo livello di sovrapposizione, utilizzando l'algoritmo [7], così da evitare gli effetti di bordo, gli spigoli netti e le interruzioni improvvise che possono presentarsi in corrispondenza del confine tra patches nella predizione, un esempio di questi effetti in Figura 2.7.

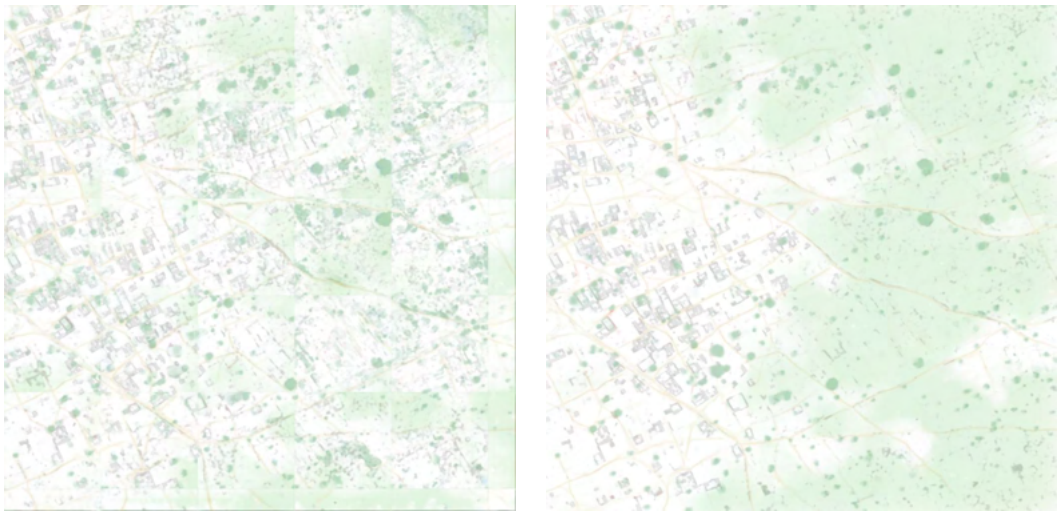

Figura 2.7: Predizione senza l'utilizzo dei blending patches sulla sinistra e con l'utilizzo dei blending patches sulla destra

2.3.1 Test

A seguire, in Figura 2.8, sono riportate alcune immagini della sezione del dataset di validazione del modello e le relative previsioni ottenute con la rete allenata.

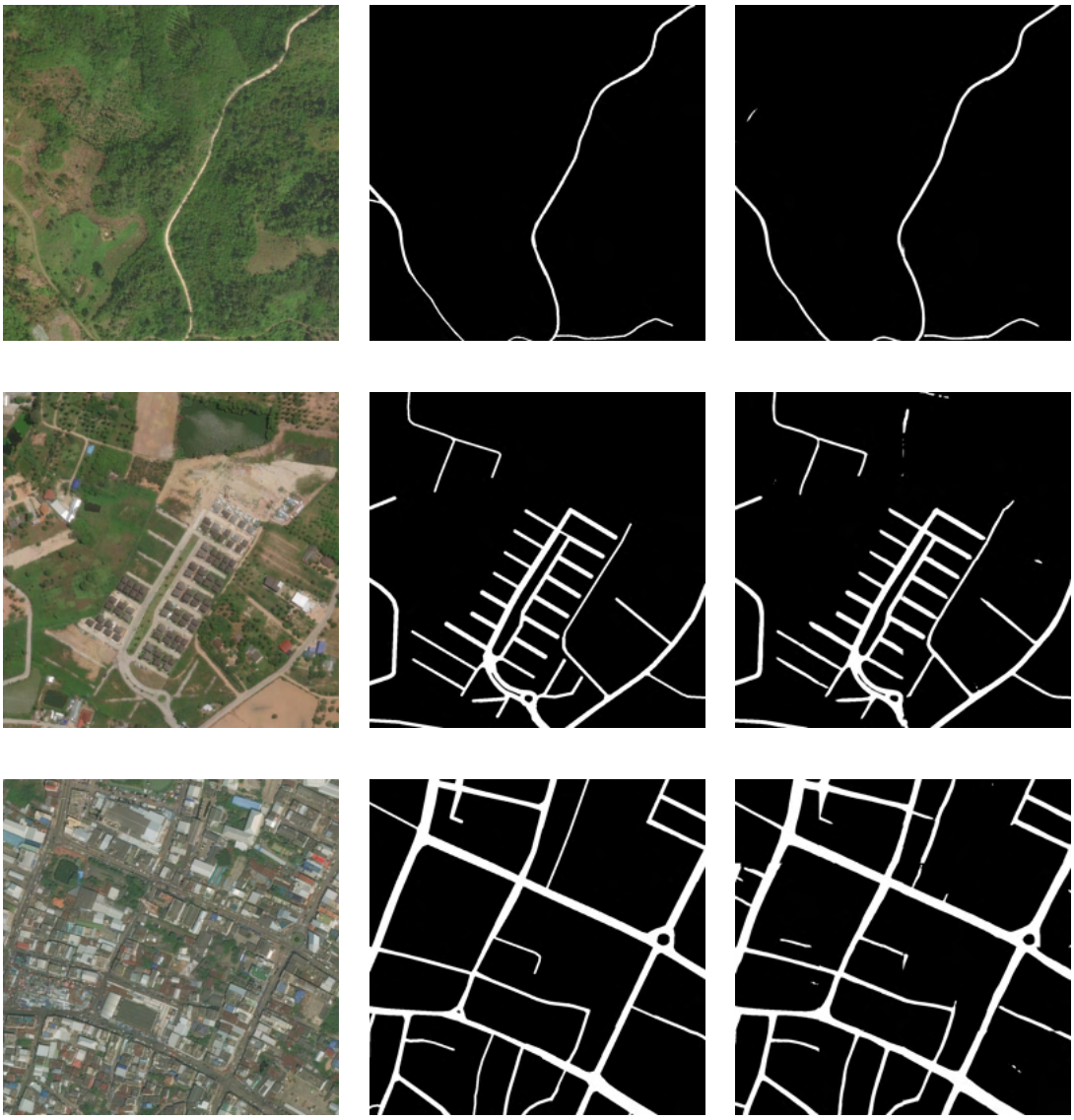

Figura 2.8: Immagine, maschera desiderata e predizione ottenuta

A dimostrazione dell'effettiva portabilità, in Figura 2.9 sono riportate alcuni previsioni relative a immagini ottenute mediante screenshot da Google Maps.

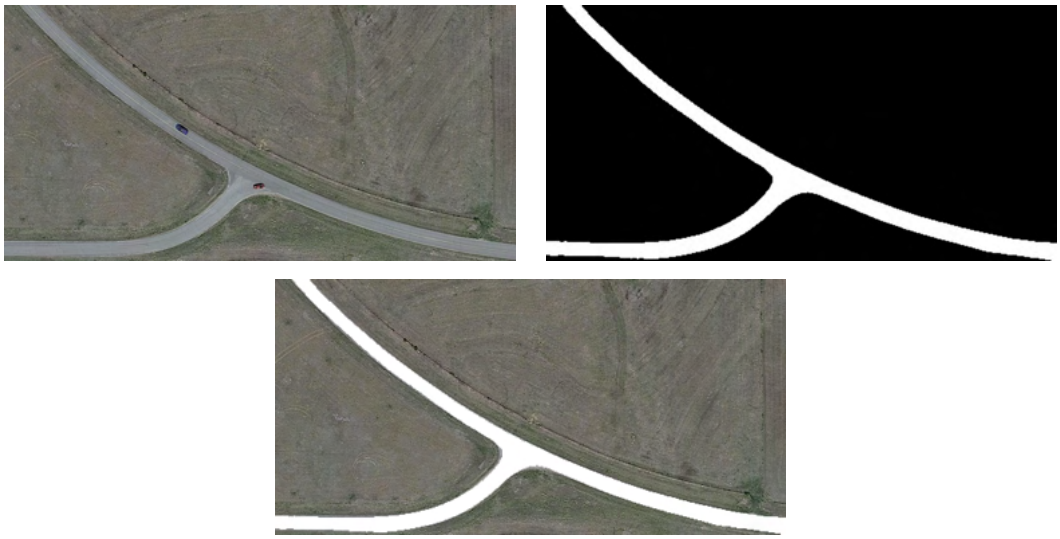

Figura 2.9: Immagine, predizione ottenuta e sovrapposizione - caso 1

Figura 2.10: Immagine, predizione ottenuta e sovrapposizione - caso 2

Figura 2.11: Immagine, predizione ottenuta e sovrapposizione - caso3

Capitolo 3

Modellazione di Gaussian Mixture Models

Questo capitolo presenta una rapida introduzione ai Gaussian Mixture Models e come sono stati integrati ed utilizzati in questa ricerca. A seguito di ciò, saranno mostrati dei risultati ottenuti combinando i GMM e l'algoritmo sviluppato per individuare le strade.

3.1 Introduzione ai GMM

Si consideri l'immagine come l'ambiente in analisi e la rete stradale come la superficie, interessata dall'evento x che si vuole analizzare con il lavoro dei robot. Risulta dunque necessario interfacciare l'intelligenza artificiale sviluppata ad un algoritmo di coverage, studiato in [8], al fine di raggiungere un copertura ottimale dell'area di interesse disponendo i droni tenendo conto delle loro posizioni relative in funzione del sensing range, raggio di visibilità, di ciascuno di questi. La soluzione proposta prevede l'utilizzo dei pixels appartenenti alle strade per modellare un Gaussian Mixture Model in grado di approssimare il più fedelmente possibile tale sistema viario. Un Gaussian Mixture Model, o GMM, è un modello probabilistico che studia tutti i punti dati come provenienti da una mixture di distribuzioni gaussiane finite. Dunque un GMM è la somma ponderata di N gaussiane che vengono mixate secondo l'equazione 3.1, tratta da [9]:

$$p(x) = \sum_{k=1}^{k=N} \omega_k g(x|\mu_k, \Sigma_k) \tag{3.1}$$

dove x è un vettore d-dimensionale e k è la k-esima gaussiana del modello. Ognuna delle gaussiane è definita da tre parametri, mostrati in Figura 3.1:

- la media, μ, definita dalle coordinate x, y e z, che ne definisce il centro;
- la matrice dxd di covarianza, Σ, che ne definisce le variazioni rispetto alla media;
- il coefficiente di combinazione, ω, che indica il peso relativo della guassiana rispetto alle altre utilizzate nel GMM.

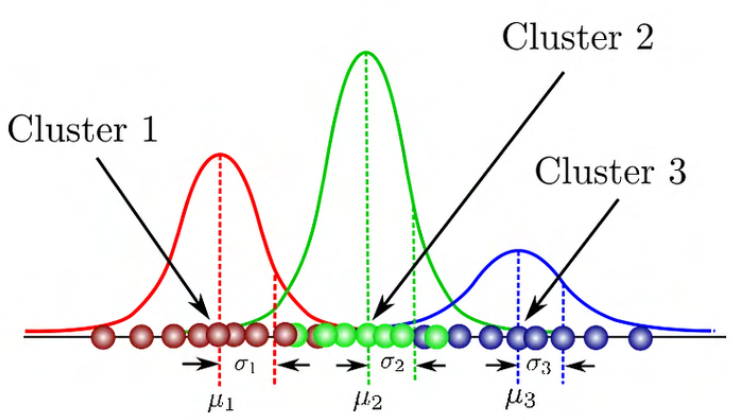

Figura 3.1: Illustrazione dei tre parametri che definiscono una gaussiana

Questi elementi vengono poi inseriti nell'espressione 3.2, introdotta in [9]

$$g(x|\mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^d |\Sigma|}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)} \tag{3.2}$$

che definisce una singola gaussiana.

3.2 Tracciamento delle Gaussiane

L'immagine da analizzare, o solo una parte di questa selezionata dall'utente tramite interfaccia grafica, viene fornita come input alla rete neurale, la quale sviluppa una predizione e comunica le coordinate, in metri, di tutti i pixel appartenenti a strade discretizzando così l'area di interesse. Ciò consente, stabilito il numero di gaussiane, la creazione del modello probabilistico il quale assegna un peso, probabilità, ai soli punti appartenenti alla regione di interesse ovvero la rete stradale. Questo è stato implementato con le seguenti righe di codice 3.2. I due cicli *for* servono per identificare tutti e i soli pixels della predizione appartenenti a strade ovvero di colore bianco, 255, così da calcolarne le coordinate in metri e fornirle alla classe che si occuperà dello sviluppo del GMM. Come si evince dal codice, la funzione *distmt* calcola le coordinate in metri di ciascun pixel effettuando una proporzione tra le coordinate x e y del pixel e le dimensioni in metri dell'immagine, ottenute con lo strumento di misurazione di Google Maps. Mentre la classe *GaussianMixture* permette di stimare i parametri ovvero punto medio, matrice di covarianza e coefficiente di combinazione di ciascuna delle gaussiane componenti il Gaussian Mixture Model. Questi parametri saranno successivamente salvati e verranno utilizzati per comandare gli agenti.

```
def distmt(x, y, img_size, mt):
    #Calcolo delle coordinate in metri dei pixels
    dist_x = (x*mt[0])/img_size[1]
    dist_y = (y*mt[1])/img_size[0]
    return [math.sqrt(dist_x**2+dist_y**2), dist_x, dist_y]

COMPONENTS_NUM = 15
IMG_SIZE = image.shape

for i in range(IMG_SIZE[0]):
    for j in range(IMG_SIZE[1]):
        if image[i,j] == 255:
            k,x,y = distmt(j, i, IMG_SIZE, DIM_MT_IMG)
            xp.append(x)
            yp.append(DIM_MT_IMG[1]-y)

GMModel = GaussianMixture(n_components=COMPONENTS_NUM,
   covariance_type='full', max_iter=1000)
GMModel.fit(np.column_stack((xp, yp)))
```

```
'''Salvataggio del punto medio, delle covarianze e del coefficiente
    di combinazione di ciascuna delle gaussiane componenti il
   modello'''
means = GMModel.means_
covariances = GMModel.covariances_
mix = GMModel.weights_
```

Esempi dello sviluppo del GMM sono riportati in Figura 3.2 e in Figura 3.3 che mostrano la predizione sviluppata, i punti della regione di interesse, ROI, la heatmap e il grafico della distribuzione di probabilità ottenuto dal tracciamento delle Gaussiane del GMM.

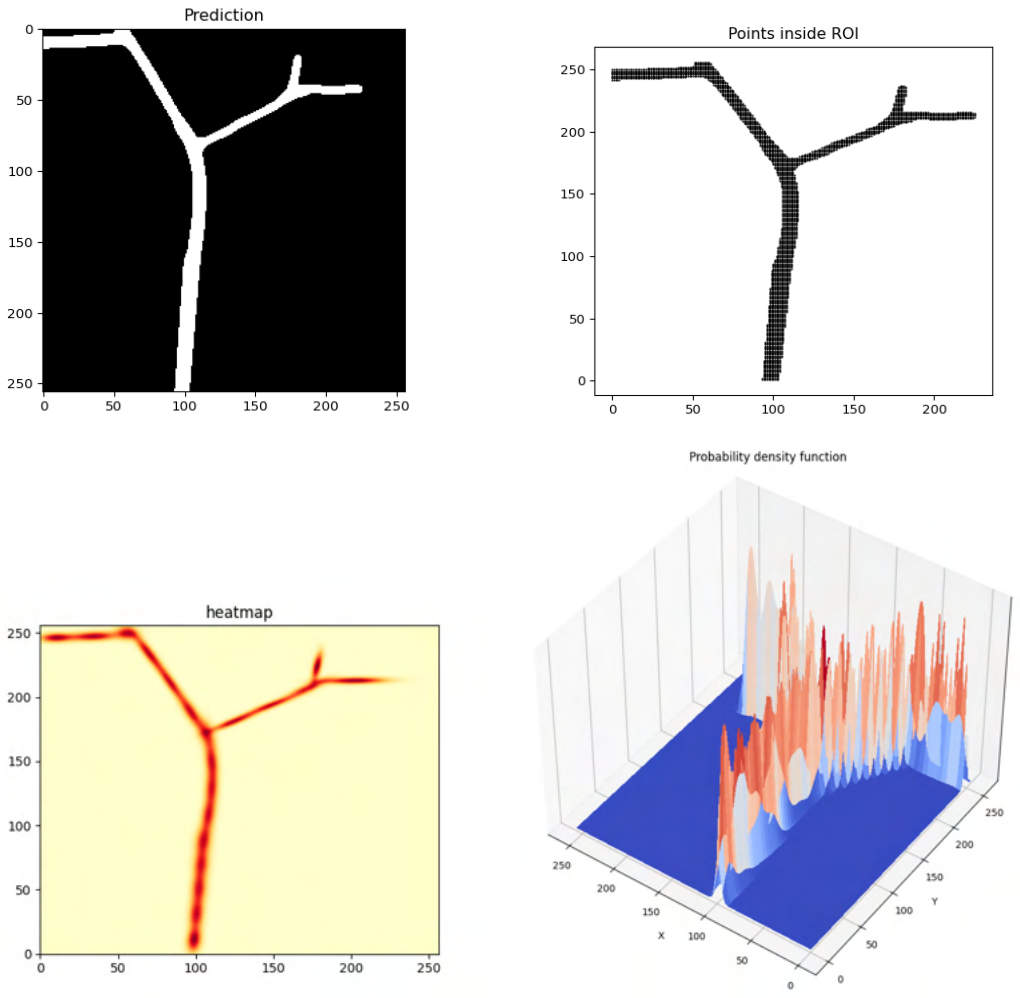

Figura 3.2: Predizione, punti ROI, heatmap e GMM - caso 1

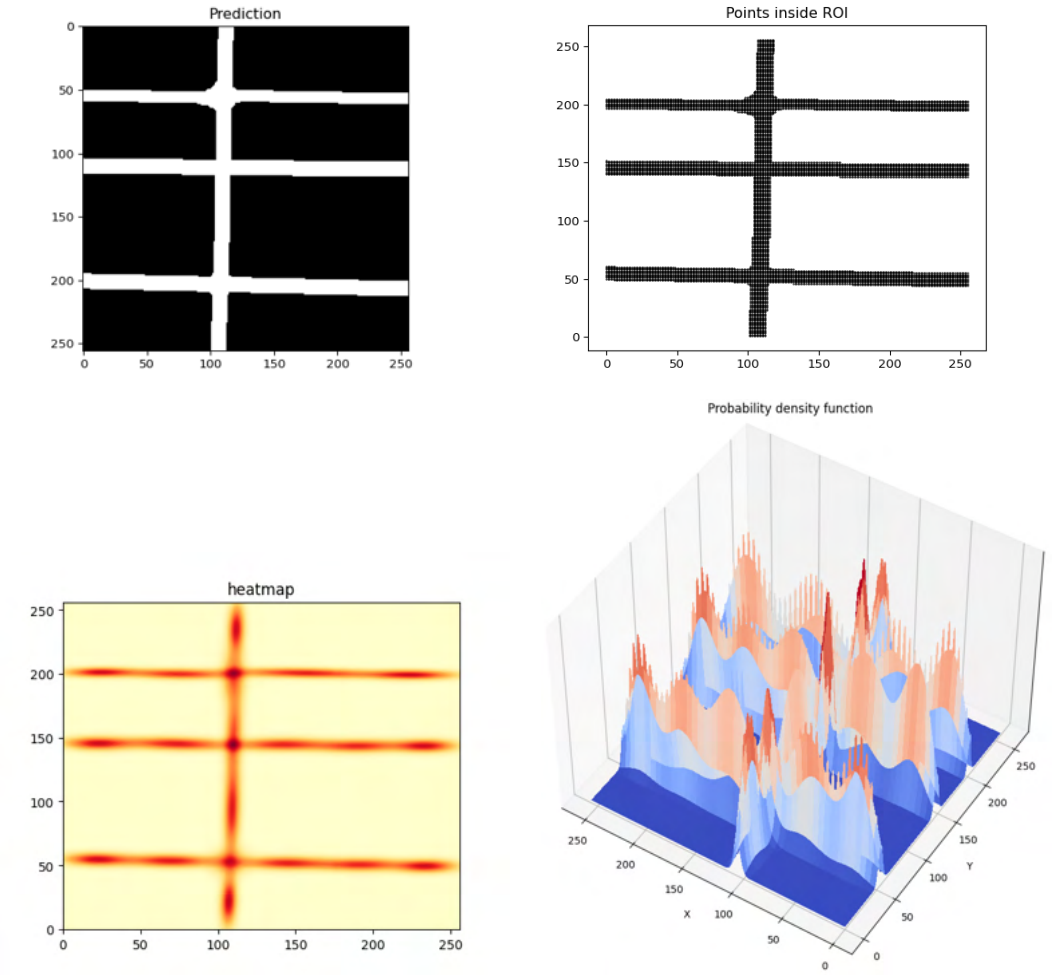

Figura 3.3: Predizione, punti di ROI, heatmap e GMM - caso 2

Capitolo 4

U-Net e GMM per il controllo di droni in ambiente ROS

In questo capitolo, successivamente ad una breve introduzione a ROS Noetic, sarà spiegato come la rete neurale e i gaussian mixture models inviano informazioni al sistema di pilotaggio dei droni. Inoltre si tratterà dell'ambiente di simulazione, Gazebo, e di come è stato possibile ricreare luoghi realistici per le sperimentazioni.

4.1 Simulazione in ROS Noetic

ROS, Robot Operating System, è un framework di sviluppo software open source per applicazioni robotiche. Un sistema ROS è composto da un numero di nodi indipendenti, ciascuno dei quali comunica con gli altri nodi utilizzando un modello di messaggistica di tipo pubblicazione/sottoscrizione. ROS Noetic inizia con il ROS Master, il quale consente a tutti gli altri nodi di comunicare tra loro. I nodi si dividono in due principali categorie: publisher, nodi che trasmettono informazioni e subscriber, che le ricevono. La comunicazione tra nodi avviene tramite la pubblicazione, e la lettura, di messaggi, che possono essere di diverse tipologie, tra cui custom, sui ROS Topics.

4.1.1 Publisher ROS

La comunicazione tra UNet, GMM e il sistema di pilotaggio dei droni è gestita da due nodi ROS. Fornita l'immagine all'intelligenza artificiale viene sviluppata una predizione, discretizzata l'area di interesse e ottenuta la funzione di distribuzione di probabilità. Il Gaussian Mixture Model così modellato, nello specifico il punto medio, la matrice delle covarianze e il coefficiente di combinazione di ciascuna delle gaussiane, compone il messaggio custom che viene pubblicato da un nodo publisher. Tale nodo è stato implementato nei seguenti passaggi 4.1.1 ed è stato realizzato organizzando il codice in funzioni. Le più rilevanti sono: la funzione *predict* che si occupa dello sviluppo della predizione fornendo lo screenshot dell'ambiente in input all'intelligenza artificiale, la funzione *gmm_model* che ricevendo come parametro la predizione computa il Gaussian Mixture Model e la funzione *create_msg* che si occupa della creazione del massaggio custom.

```
def create_msg(mns, cvs, mix):
    #creazione del messaggio custom per la pubblicazione del gmm
    gmm_msg = GMM()
    for i in range(len(mns)):
        g = Gaussian()
        mean_pt = Point()

        mean_pt.x = mns[i][0]
        mean_pt.y = mns[i][1]
        mean_pt.z = 0.0
        g.mean_point = mean_pt
        for j in range(len(cvs[i])):
            g.covariance.append(cvs[i][j][0])
            g.covariance.append(cvs[i][j][1])

        gmm_msg.gaussians.append(g)
        gmm_msg.weights.append(mix[i])
    return gmm_msg

if __name__ == '__main__':
    rospy.init_node("gmm_node")
    rospy.loginfo("Node has been started")

    pub = rospy.Publisher("/gaussian_mixture_model", GMM,
       queue_size=10)
```

```
#Scenario 1
image_path = '/home/ubuntu/env4.png'
DIM_MT_IMG = [240, 106]

#predict: funzione creata per effettuare la predizione
prediction = predict(image_path, model_path)

#Calcolo delle dimensioni reali dell'ambiente in funzione alla
   predizione ridimensionata
DIM_MT_IMG = get_real_mt(DIM_MT_IMG, image_path)

#gmm_model: funzione creata per sviluppare il GMM
mns, cov, mix = gmm_model(prediction, DIM_MT_IMG)

rate = rospy.Rate(1)

while not rospy.is_shutdown():
    #Creazione del messaggio
    msg = create_msg(mns, cov, mix)

    #Pubblicazione del messaggio
    pub.publish(msg)

    rate.sleep()
```

Le informazioni pubblicate vengono successivamente lette da un nodo subscriber, gmm-coverage, il quale controlla in velocità i droni affinché si dispongano in modo ottimale rispetto alla rete stradale dell'ambiente di simulazione. Il controllo avviene calcolando i baricentri delle celle di Voronoi costruite attorno agli agenti stessi per definire la loro nuova posizione. Il centroide di ciascuna cella è individuato attraverso la media ponderata calcolata utilizzando le coordinate dei punti e i relativi pesi, valori della funzione densità di probabilità. La visualizzazione dello spostamento dei droni all'interno dell'ambiente di simulazione avviene grazie a Gazebo[1].

[1]Gazebo è un simulatore di robotica 3D open source, fornisce un rendering realistico degli ambienti, tra cui illuminazione, ombre e trame di alta qualità.

4.1.2 Blender e RenderDOC

La creazione di ambienti di simulazione reali in Gazebo è stata resa possibile grazie all'utilizzo di RenderDOC[2], Blender[3] e Google Earth. RenderDOC consente di tradurre ciò che la GPU carica in un certo istante, scelto dall'utente, da Google Earth, in un modello 3D importabile su Blender grazie ad un add-on [10]. Quest'ultimo è stato utilizzato per convertire tale modello in un formato compatibile con Gazebo e salvare le textures dell'ambiente e degli edifici presenti nell'area catturata. Alla fine di questo processo si ottengono un ambiente virtuale fedele alla realtà e le sue textures, che se collocati in opportune cartelle e richiamati nel launch file consentono la visualizzazione di tale ambiente in Gazebo, Figura 4.1.
Per la simulazione dei droni nell'ambiente virtuale è stato scelto il modello mostrato in Figura 4.2 fedele all'hardware disponibile in laboratorio e sono stati utilizzati i pacchetti [11] [12] [13].

Figura 4.1: Roma, piazza del Colosseo, ambiente di simulazione

[2]RenderDoc è un debugger grafico autonomo che consente l'acquisizione, rapida e semplice, e l'introspezione dettagliata di fotogrammi.

[3]Blender è un set di strumenti software di computer grafica 3D open source utilizzato per creare tra le tante cose film animati, effetti visivi e modelli stampati in 3D.

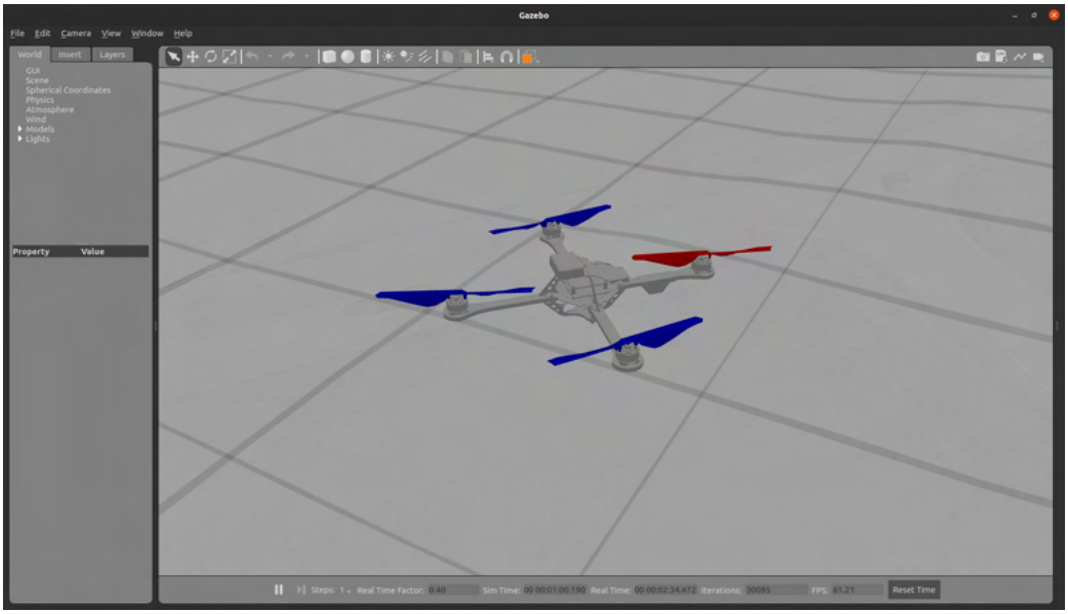

Figura 4.2: Modello Firefly dei droni in Gazebo

Capitolo 5

Risultati

Questo capitolo presenta i risultati ottenuti applicando la rete neurale e le tecniche di coverage presentate al controllo di agenti. Saranno dunque mostrate la simulazioni su diversi ambienti virtuali per valutare le prestazioni ottenute della soluzione proposta e per testare in un ambiente sicuro il controllo degli agenti.

5.1 Simulazioni

Le posizioni di partenza dei droni sono state scelte arbitrariamente in quanto è stato provato che non influenzano la capacità dei droni di assumere la distribuzione desiderata. Il sensing range è stato impostato a 25 metri. Questo ha permesso ai droni di disporsi equamente nella regione di interesse tenendo in considerazione la posizione degli altri agenti nel sensing range.
L'algoritmo è stato testato in due differenti scenari, nel primo caso l'area scelta consiste nell'intero ambiente di simulazione, Figura 5.1. Sono rappresentati in Figura 5.2 e in Figura 5.3, rispettivamente la posizione iniziale e finale dei droni. La Figura 5.4 mostra il diagramma di Voronoi, in cui sono rappresentati gli agenti nelle posizioni finali, in verde, la partizione dello spazio tra gli agenti e i punti medi delle gaussiane componenti il GMM, in giallo. Questo valida ultimamente la possibilità di applicare l'algoritmo proposto al problema studiato.

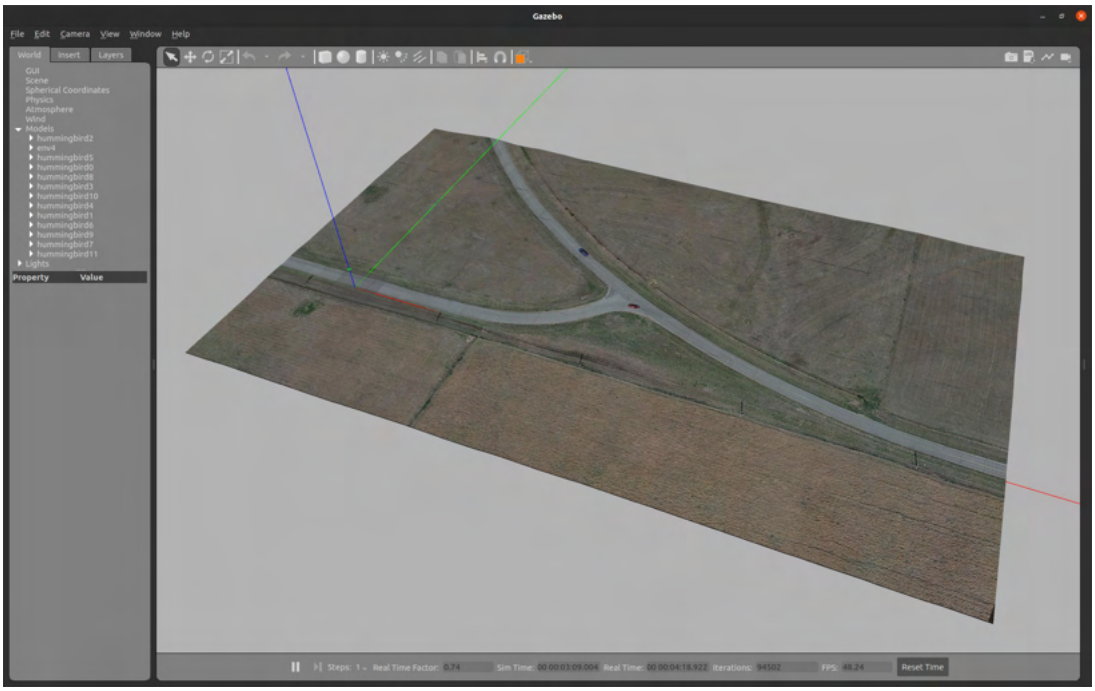

Figura 5.1: Ambiente di simulazione - Scenario 1

Figura 5.2: Disposizione iniziale dei droni, arbitraria - Scenario 1

Figura 5.3: Posizioni finali raggiunte dai droni - Scenario 1

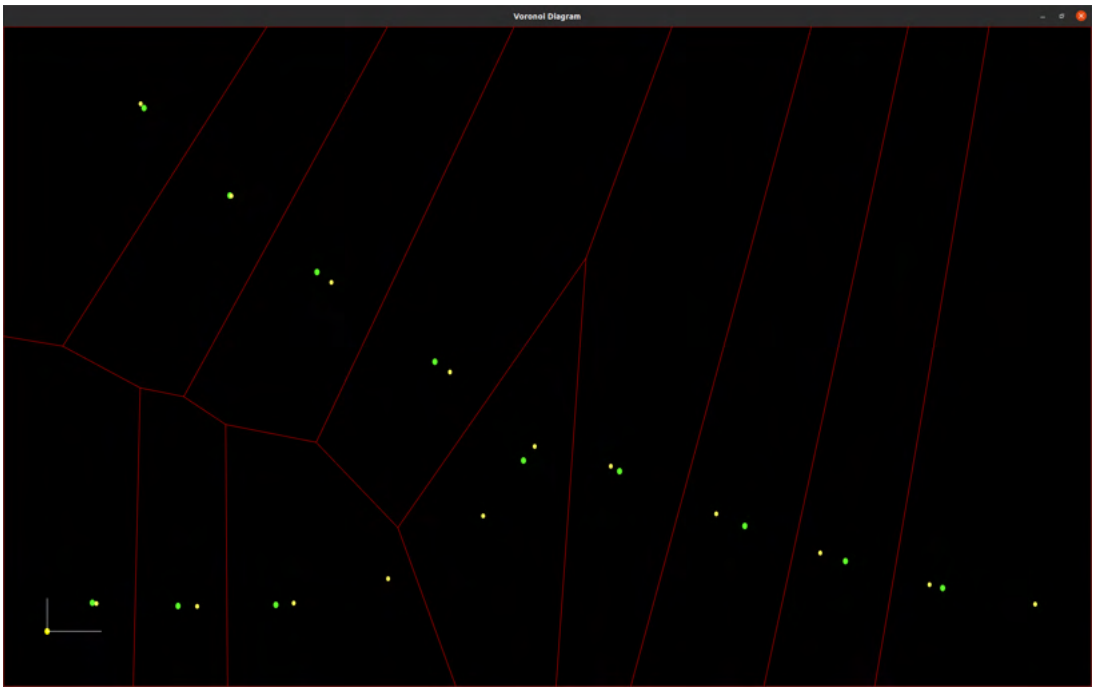

Figura 5.4: Diagramma di Voronoi, i punti gialli sono i punti medi delle gaussiane mentre i verdi i droni nella loro configurazione finale - Scenario 1

La seconda simulazione è stata effettuata su una porzione dell'ambiente selezionata dall'utente attraverso interfaccia grafica, selezione in Figura 5.5. Questa modalità offre la possibilità di disporre gli agenti unicamente in una regione specifica. L'am-

biente in cui si è simulato lo spostamento dei droni è mostrato in Figura 4.1, la posizione iniziale degli agenti è visibile in Figura 5.6 e quella finale in Figura 5.7. Anche in questo caso, dal diagramma di Voronoi, in Figura 5.8, si evince il corretto funzionamento dell'algoritmo in quanto gli agenti raggiungono le posizioni all'interno dell'area selezionata che garantiscono il miglior coverage di questa.

Figura 5.5: Immagine dell'ambiente come canvas su cui selezionare la regione desiderata - Scenario 2

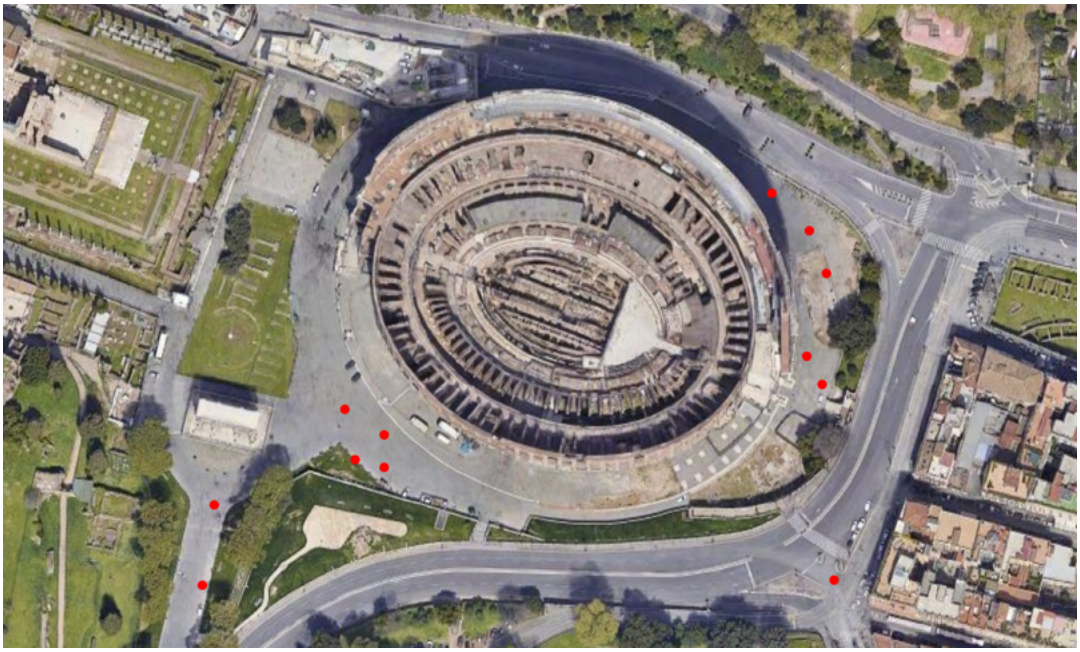

Figura 5.6: Disposizione iniziale dei droni, arbitraria - Scenario 2

Figura 5.7: Posizioni finali raggiunte dai droni - Scenario 2

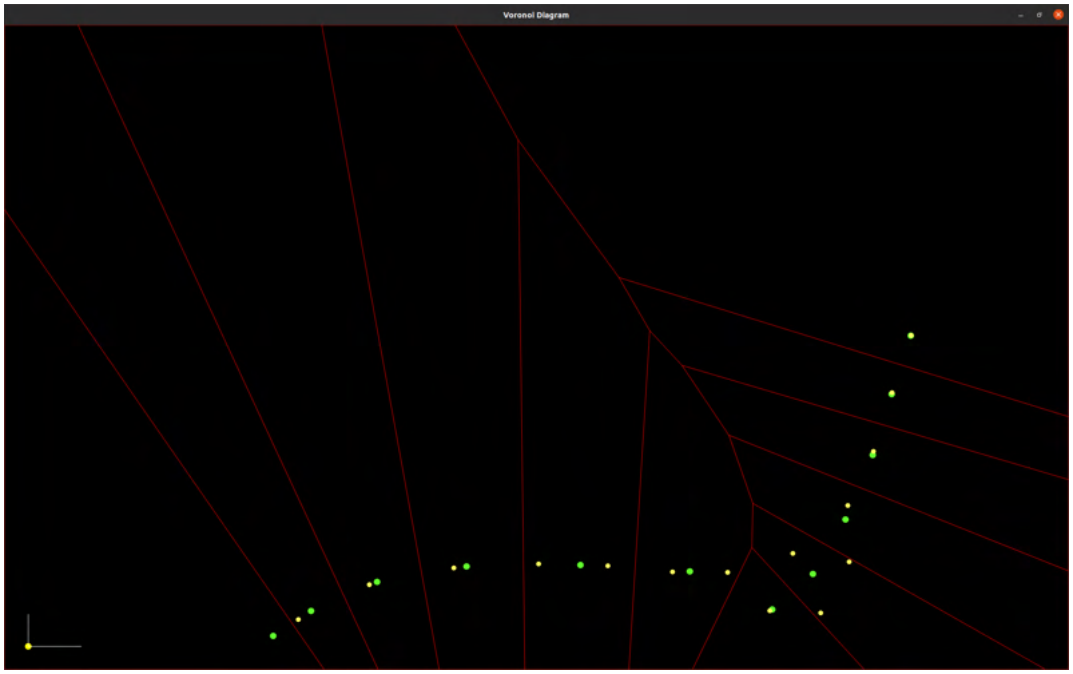

Figura 5.8: Diagramma di Voronoi, i punti gialli sono i punti medi delle gaussiane mentre i verdi i droni nella loro configurazione finale - Scenario 2

Capitolo 6

Conclusione e Sviluppi Futuri

A conclusione del lavoro svolto è possibile fare un bilancio dei risultati ottenuti. Con l'intento di proporre una soluzione all'ottenimento del coverage della rete stradale mediante un sistema multi-agente, si è sviluppato un algoritmo per il riconoscimento del sistema viario che si interfacciasse, mediante il calcolo di un GMM, con l'algoritmo di coverage per creare una tassellazione ponderata dello spazio per assegnare le posizioni ottimali agli agenti. Per l'estrazione della rete stradale da una foto, si è ricorso a una rete neurale basata su una architettura U-Net. Successivamente si è interfacciata tale rete neurale con un algoritmo capace di sviluppare un Gaussian Mixture Model per il calcolo della funzione di densità di probabilità dell'evento. Infine, la creazione di due nodi ROS per il controllo dei droni ha reso possibile simulare in Gazebo il moto degli agenti verso le rispettive posizioni finali garantendo così il coverage ottimale della rete stradale. Dagli esiti delle simulazioni emerge un ottimo livello di accuratezza, che conferma il corretto funzionamento dell'algoritmo sviluppato per l'identificazione della rete stradale, dell'interfaccia tra questo e l'algoritmo di coverage e l'adeguatezza per il compito analizzato.
Dati i risultati ottenuti in simulazione, potrebbe essere interessante testare l'algoritmo in ambienti reali. Ciò consentirebbe di studiare la sua risposta in un ambiente che presenta imprevisti e hardware. Un possibile sviluppo futuro potrebbe essere quello di implementare una intelligenza artificiale capace di riconoscere le case, le proprietà private e gli alberi con lo scopo di comunicare agli agenti la presenza di ostacoli nell'ambiente. Complicazione che attualmente è risolta elevando i droni ad

un'altezza sufficiente per evitare tali ostacoli. Questo upgrade renderebbe l'algoritmo utilizzabile con agenti diversi da droni, ad esempio robot a terra, estendendone così il campo di utilizzo anche alle "Drone No-Fly Zones". Un altro eventuale miglioramento da apportare è la conversione dell'algoritmo di coverage, attualmente centralized, i calcoli sono svolti da un calcolatore che comunica i risultati agli agenti, a distributed, ciascun agente svolge i calcoli per determinare la posizione da raggiungere, così da rendere gli agenti indipendenti da un calcolatore centrale. Questo consentirebbe un aumento della versatilità e della portabilità del sistema.
Sarebbe interessante valutare i vantaggi che porta l'algoritmo in altri scenari, un esempio è il controllo delle masse d'acqua, come laghi, corsi d'acqua o oceani, per la rimozioni di rifiuti solidi galleggianti, ad esempio plastica. In questo scenario la disposizione ottimale degli agenti assicurerebbe una pronta rimozione dell'inquinante prevenendo la contaminazione dell'ecosistema. Per questa applicazione l'intelligenza artificiale oltre a garantire il pattugliamento della sola massa d'acqua garantirebbe anche l'identificazione del rifiuto e dunque il suo smaltimento.
Con l'implementazione e l'integrazione di questi accorgimenti è auspicabile che in futuro si raggiunga un livello di qualità tale da permettere l'utilizzo del codice per missioni sul campo.

Ringraziamenti

"Macte nova virtute, puer, sic itur ad astra"
Virgilio, Eneide, IX

Considero doveroso prima di tutto ringraziare il professor Lorenzo Sabattini che mi ha consentito di lavorare ad un progetto stimolante in un ambito innovativo che considero affascinante.
Vorrei esprimere la mia gratitudine anche verso il correlatore di questa tesi, Mattia Catellani, per l'assistenza offerta durante lo svolgimento di questa ricerca.
Inoltre vorrei ringraziare mia sorella per l'illimitata pazienza che ha nei miei confronti e per la sua disponibilità.
Infine uno speciale ringraziamento va a Matilde; grazie per esserci sempre stata, per avermi supportato e sopportato per tutti questi anni.

Bibliografia

[1] **Olaf Ronneberger, Philipp Fischer e Thomas Brox**. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. arXiv: 1505.04597 [cs.CV].

[2] **Ilke Demir et al.** «DeepGlobe 2018: A Challenge to Parse the Earth Through Satellite Images». In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*. Giu. 2018.

[3] **Fausto Milletari, Navab Nassir e Ahmadi Seyed-Ahmad**. «V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation». In: (2016).

[4] **Metika Sikka**. *Balancing the Regularization Effect of Data Augmentation*. https://towardsdatascience.com/balancing-the-regularization-effect-of-data-augmentation-eb551be48374. 2020.

[5] **Diederik P. Kingma e Jimmy Ba**. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[6] **Sreenivas Bhattiprolu**. *Test Time Augmentation for Semantic Segmentation*. https://github.com/bnsreenu/python_for_microscopists. 2021.

[7] **Guillaume Chevalier**. *Smoothly Blend Image Patches*. https://github.com/Vooban/Smoothly-Blend-Image-Patches. 2017.

[8] **Mattia Catellani et al.** «Distributed Control for Human-Swarm Interaction In Non-Convex Environments using Gaussian Mixture Models». In: *IfacPapersOnline* (2023).

[9] **Douglas A. Reynolds**. «Gaussian Mixture Models». In: *Encyclopedia of Biometrics*. 2009.

[10] **Elie Michel**. *Maps Models Importer*. `https://github.com/eliemichel/MapsModelsImporter`. 2023.

[11] **Autonomous Systems Lab @ ETH Zürichl**. *Rotors Simulator*. `https://github.com/ethz-asl/rotors_simulator`. 2021.

[12] **Robotics e Perception Groupl**. *RPG Quadrotor Control*. `https://github.com/uzh-rpg/rpg_quadrotor_control`. 2021.

[13] **Robotics e Perception Groupl**. *RPG Quadrotor Common*. `https://github.com/uzh-rpg/rpg_quadrotor_common`. 2021.