SAPIENZA
UNIVERSITÀ DI ROMA

Anomaly detection su uva mediante segmentation e open-set object detection

Facoltà di Ingegneria dell'informazione, informatica e statistica

Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Giovanni Zara
Matricola 1929181

Relatore

Prof. Thomas Alessandro Ciarfuglia

Anno Accademico 2023/2024

Tesi non ancora discussa

Anomaly detection su uva mediante segmentation e open-set object detection
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: giovizara@gmail.com

Sommario

Il presente studio propone lo sviluppo di algoritmi per la risoluzione di problemi di anomaly detection, su un dataset composto da immagini di uva da tavola. Nello specifico, l'obiettivo della ricerca è quello di sviluppare un sistema automatizzato che identifichi anomalie all'interno di grappoli d'uva, mediante l'impiego di reti neurali e varie tecniche di pre e post processing dei dati.

La trattazione è divisa in:

- un primo stadio incentrato sulla preparazione e analisi dei dati;
- una sezione intermedia che si focalizza sulla segmentazione e sull'estrazione delle features dalle immagini;
- la conclusione finale di effettiva classificazione dell'uva e calcolo delle metriche di valutazione.

Particolare rilievo riveste il lavoro di segmentazione e identificazione di oggetti, per il quale è stato utilizzato GSAM *(Grounded Segment Anything)*, un modello che unisce l'attuale stato dell'arte della image segmentation *(Segment Anything)* con quello della open-set object detection *(Grounding Dino)*. Attraverso questo modello, che risolve problemi di zero shot learning, è stato possibile pre-processare i dati isolando le porzioni più significative di essi, prima identificando l'oggetto (GD) e poi segmentandolo (SAM).
È stata successivamente eseguita l'estrazione delle features dalle maschere di immagine con una CNN *(Convolutional Neural Network)*, una rete neurale convoluzionale, con le quali è stata poi allenata una seconda semplice rete neurale per operare la classificazione.

Il presente lavoro rientra nel progetto europeo "CANOPIES", che mira a sviluppare un nuovo paradigma collaborativo uomo-robot nel campo dell'agricoltura di precisione per colture permanenti, in cui i lavoratori agricoli possano collaborare in modo efficiente con gruppi di robot per eseguire interventi agronomici di vario genere.

Indice

Capitolo 1

Introduzione

1.1 Descrizione del task

Il problema che questa ricerca propone di affrontare è quello dell'anomaly detection, ossia l'individuazione di anomalie, eventi o punti chiave che deviano da ciò che è definito standard, rendendo determinati dati inconsistenti rispetto alla totalità del dataset. Distinguere un dato anomalo da uno standard è un tema centrale in numerosi studi e ricerche nel campo del machine learning, e rappresenta uno degli utilizzi più promettenti dell'intelligenza artificiale. Alcuni algoritmi sono sviluppati per rispondere a esigenze specifiche, come nel caso di questa tesi, mentre altri sono progettati per scopi più generici.

I problemi di anomaly detection non sono in generale facili da risolvere.

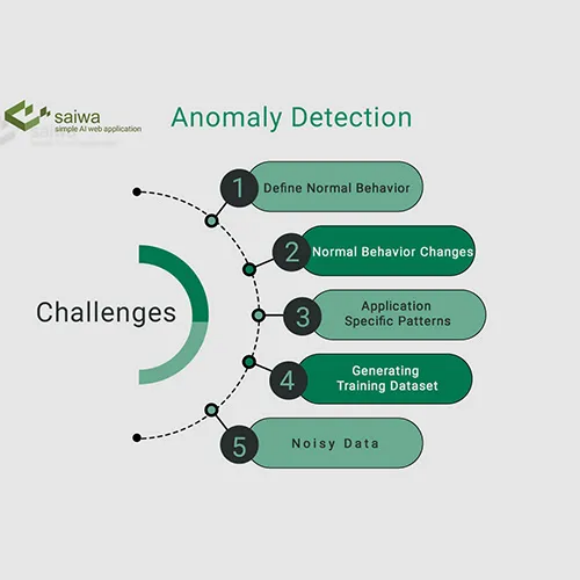

Figura 1.1. anomaly detection steps. Fonte: [1]

Una delle sfide più complesse risiede nel definire un intervallo che includa tutti i possibili comportamenti "normali" di un certo dato. Il confine tra normale ed anomalo è spesso molto labile, rendendo sempre più complessa l'individuazione degli outliers. Inoltre in molti campi, a seconda del dominio di applicazione, il comportamento standard è in continua evoluzione, perciò una definizione attuale di normalità potrebbe risultare insufficiente se applicata a dati futuri.

Infine, pur avendo definito confini di valutazione robusti, ogni dato continuerà sempre a contenere una quantità minima di "rumore", che risulta spesso difficile da individuare ed eliminare.

Quando ci si approccia ad un problema di anomaly detection, si può procedere sviluppando algoritmi ad apprendimento supervisionato o non supervisionato.

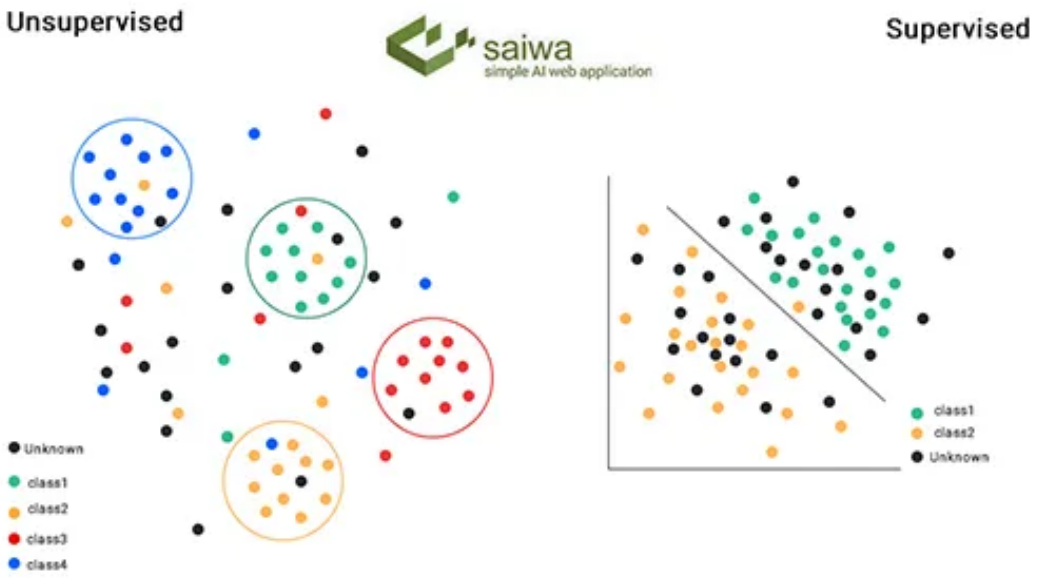

Figura 1.2. Apprendimento supervisionato e non supervisionato. Fonte: [2]

Nell'apprendimento supervisionato (utilizzato nel presente studio) gli elementi del dataset sono tutti classificati in maniera binaria, come normali o anomali. Il modello sfrutta questa suddivisione per estrarre caratteristiche dalle differenti immagini e individuare vari pattern. Per questo tipo di studio è essenziale quindi che i dati utilizzati dal modello durante la fase di training siano di alta qualità, altrimenti le prestazioni complessive potrebbero risultare compromesse.

Per quanto riguarda invece l'apprendimento non supervisionato, il modello lavora su dati che non conosce e non precedentemente classificati. Di norma, una rete neurale analizza i dati, li classifica e identifica vari pattern di anomalie. Questo approccio consente al modello di ridurre significativamente la necessità di pre-processamento manuale dei dati, rendendolo più autonomo e migliorandone la scalabilità.

Capitolo 2

Dataset

2.1 Classe del dataset

Il dataset utilizzato per questo studio è costituito da una serie di immagini di uva da tavola, preliminarmente divise in sane e anomale con diverse scale di definizione. Le immagini sono inserite in un file system gerarchico per suddividere i due tipi e le varie definizioni. I dati sono stati organizzati e gestiti mediante un'apposita classe creata con un modulo del framework pytorch. Le immagini sono state poi raggruppate solo per tipo ignorando le scale di definizione, evitando quindi di creare differenze di trattamento delle stesse, potenzialmente dannose per il bilanciamento del dataset se non pesate opportunamente. Di seguito un estratto del codice raffigurante la classe utilizzata per la gestione del dataset:

Listing 2.1. classe per la creazione del dataset

```
class CanopiesDataset(Dataset):
    def __init__(self, root_dir, classe="every", scale="every"):
        self.root_dir = root_dir
        self.class_names = sorted(os.listdir(root_dir), reverse=True)
        self.image_paths = []

        for class_name in self.class_names:
            class_dir = os.path.join(root_dir, class_name)
            scale_dirs = os.listdir(class_dir)
            for scale_dir in scale_dirs:
                scale_path = os.path.join(class_dir, scale_dir)
                image_names = os.listdir(scale_path)
                for image_name in image_names:
                    if image_name.endswith(".jpg"):
                        if "hdr" not in image_name.lower():
                            if (classe == "every" or classe ==
                                class_name):
```

```
17                                  if (scale == "every" or scale ==
                                        scale_dir):
18                                      image_path = os.path.join(
                                            scale_path, image_name)
19                                      self.image_paths.append((
                                            image_path, self.class_names.
                                            index(class_name)))
```

2.2 Analisi esplorativa dei dati (EDA)

2.2.1 Campioni sani

Di seguito un esempio esplorativo di campioni di uva sana.

Figura 2.1. campioni sani

2.2.2 Campioni anomali

Di seguito un esempio esplorativo di campioni di uva anomala.

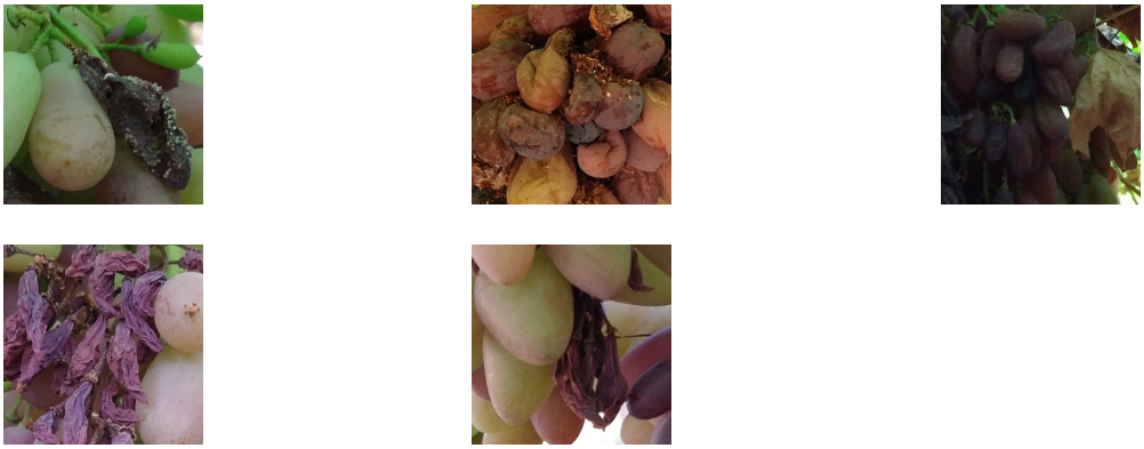

Figura 2.2. campioni malati

2.3 Divisione del dataset e bilanciamento

Il dataset presenta uno sbilanciamento tra la quantità di immagini sane e anomale.

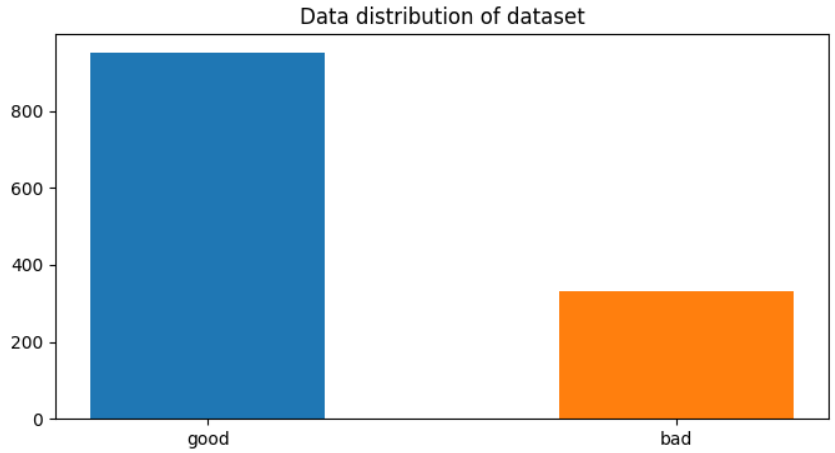

Figura 2.3. distribuzione

Esiste un criterio, chiamato imbalance rateo (rateo di sbilanciamento), che permette di valutare in maniera più o meno approssimativa quanto l'insieme di dati di cui si dispone sia sbilanciato:

$$ImbalanceRatio = \frac{NumberOfSamplesInMinorityClass}{NumberOfSamplesInMajorityClass} = \frac{350}{950} \approx 0.3684 \quad (2.1)$$

Per problemi di anomaly detection come questo generalmente viene tollerato un rateo di sbilanciamento di circa 1/3, dove per ogni elemento nella classe di minoranza (anomalie) si hanno 3 elementi della classe di maggioranza (uva sana). Essendo il rateo di circa 0.36, posso considerare il dataset accettabile senza eseguire operazioni di sottocampionamento (che eventualmente verranno effettuate nella fase di estrazione delle features)

L'intero dataset è stato diviso in una porzione dedicata al training, una alla validation ed una al testing dei dati. Ogni frazione è stata utilizzata in modo coerente in ogni operazione effettuata all'interno della ricerca, per mantenere la coesione e l'interdipendenza dei risultati e non far perdere di significato alle metriche di valutazione.

Il training set (60%) rappresenta quell'insieme di dati che viene utilizzato per insegnare al modello le nozioni di base di cui necessita per individuare le anomalie; il validation set (20%) serve a validare il modello, identificando la combinazione migliore di iperparametri che permette un apprendimento ottimale; infine, nel test set (20%) figurano i dati che vengono utilizzati per testare il funzionamento e le

prestazioni del modello, i quali, per scongiurare l'overfitting, devono essere dati che l'algoritmo non ha mai analizzato precedentemente.

Ciò vuol dire che ogni operazione viene effettuata separatamente su ogni tipo di dato, processandone il risultato all'interno della stessa partizione. Questo per evitare, ad esempio, che dati di train dopo un post-processing vengano poi divisi in maniera differente, terminando in una porzione non dedicata più al train.

Di seguito uno snippet del codice delle funzioni utilizzate per la divisione dei dati:

Listing 2.2. funzioni di splitting

```
1  def split_dataset(dataset, train_ratio=0.7, val_ratio=0.2, test_ratio
       =0.1):
2      epsilon = 1e-10
3      assert abs(train_ratio + val_ratio + test_ratio - 1.0) < epsilon,
           "Ratios must sum to 1"
4      dataset_size = len(dataset)
5      indices = list(range(dataset_size))
6      random.shuffle(indices)
7      train_end = int(train_ratio * dataset_size)
8      val_end = train_end + int(val_ratio * dataset_size)
9      train_indices = indices[:train_end]
10     val_indices = indices[train_end:val_end]
11     test_indices = indices[val_end:]
12     train_subset = Subset(dataset, train_indices)
13     val_subset = Subset(dataset, val_indices)
14     test_subset = Subset(dataset, test_indices)
15     return train_subset, val_subset, test_subset
16 def create_dataloaders(train_subset, val_subset, test_subset,
       batch_size=32):
17     train_loader = DataLoader(train_subset, batch_size=batch_size,
           shuffle=True)
18     val_loader = DataLoader(val_subset, batch_size=batch_size,
           shuffle=False)
19     test_loader = DataLoader(test_subset, batch_size=batch_size,
           shuffle=False)
20     return train_loader, val_loader, test_loader
```

Capitolo 3

Grounded Segment Anything

3.1 La tecnologia alla base

Prima di analizzare il framework completo, un rapido approfondimento sulle tecnologie alla base di quest'ultimo.
Grounded Segment Anything combina le capacità di zero-shot object detection di Grounding Dino con quelle di image segmentation di Segment Anything.

3.1.1 Grounding Dino (GD)

Grounding Dino è uno zero-shot detector, in grado di classificare e riconoscere oggetti mai visti durante le fasi di allenamento a partire da un input di testo. Utilizza DINO (Distilled Knowledge from Internet pre-trained models) per interpretare prompt testuali e generare precise label e bounding boxes per oggetti all'interno di immagini. La backbone del modello è costituita da Vision Transformers (ViTs), allenati su grandi dataset di immagini non classificate per imparare varie rappresentazioni visuali. Un vision transformer (ViT) è un transformer designato per la computer vision: divide l'immagine in una serie di patches (piuttosto che di token, a differenza di un transformer classico per utilizzo testuale) e serializza ogni patch in un vettore per mapparlo poi su uno di dimensioni ridotte con una singola moltiplicazione matriciale. Questi vettori sono poi processati da un encoder, per trovare una corrispondenza testuale. Quindi, senza un'esposizione a classi di oggetti specifiche, Grounding Dino è in grado di comprendere prompt testuali e localizzarli nell'immagine. Riconosce oggetti con un'ottima generalizzazione utilizzando un modello intuitivo di riconoscimento testuale, colmando il divario tra linguaggio e percezione visiva.

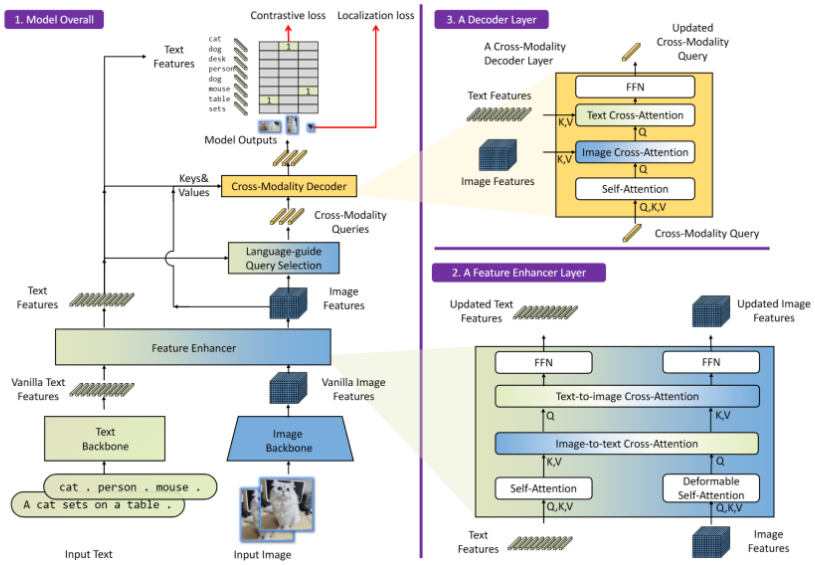

Figura 3.1. Il modello: text backbone, image backbone, feature enhancer, language-guided query selection, e cross-modality decoder. Fonte : [3]

3.1.2 Segment Anything (SAM)

Segment Anything (SAM) è un foundation model, capace di segmentare ogni entità distinguibile all'interno di un'immagine generandone una maschera.

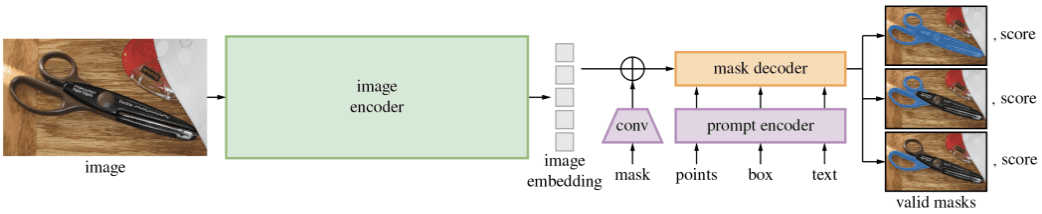

Figura 3.2. model diagram. Fonte: [4]

Rappresenta lo stato dell'arte della segmentazione di immagini, utilizza i principi del few-shot learning e i ViTs, adattandosi ad una gamma di task molto ampia. Le capacità rivoluzionarie del modello sono frutto della sua architettura: un image decoder, un prompt decoder e un mask decoder.

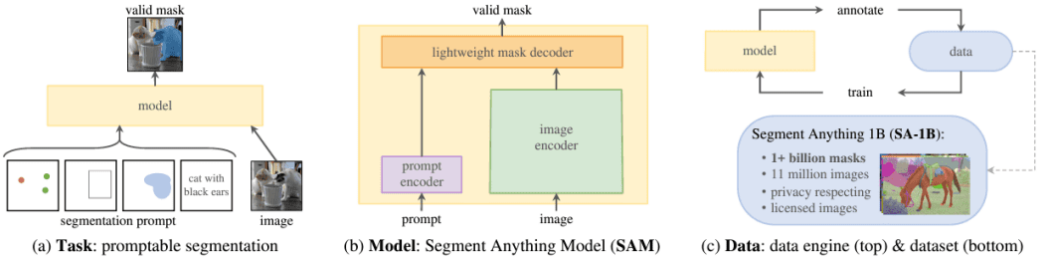

Figura 3.3. Architettura del modello SAM. Fonte: [5]

- Image encoder: è il cuore dell'architettura, la componente responsabile della trasformazione dell'input in un insieme di features.
- Prompt encoder: differenzia SAM dai modelli di segmentazione tradizionali. Interpreta varie combinazioni di prompt (nel caso di questo progetto principalmente bounding boxes) traducendole in suggerimenti che guidano il processo di segmentazione.
- Mask decoder: sintetizza le informazioni provenienti dai due componenti precedenti per produrre accurate maschere di immagine.

L'interazione efficace di queste tre componenti dà vita alla segmentazione. L'image encoder comprende accuratamente l'intera immagine, dividendola in piccoli frammenti analizzabili dal modello. Comprime le immagini in una densa matrice di features, che forma la base di conoscenza che il modello utilizza per identificare elementi dall'immagine (con un approccio basato sui transfomers, similmente a ciò che accade per i modelli di natural language processing). Il prompt encoder interpreta i vari prompt forniti al modello, spostando la concentrazione dell'analisi su determinate aree o oggetti. Infine, il mask decoder usa una combinazione delle informazioni precedenti per segmentare effettivamente l'immagine, assicurandosi che l'output sia allineato con il prompt iniziale.

La backbone di SAM è costituita da un'unione di Convolutional Neural Networks (CNNs) e Generative Adversarial Networks (GANs). Questi modelli di deep learning sono centrali nello sviluppo dell'image processing nel campo del'AI e forniscono la base che rende SAM lo strumento di segmentazione più sofisticato al momento. Le CNNs eccellono nel riconoscimento di schemi e pattern dalle immagini, imparando strutture spaziali gerarchiche di features, partendo da semplici angoli fino ad arrivare a forme geometriche complesse. Il contributo delle GANs invece risiede nella loro

abilità di generare maschere d'immagine precise, facendo affidamento sulle capacità del generatore e del discriminatore. La generazione produce immagini ad alta verosimiglianza mentre il discriminatore determina se queste ultime sono reali o artificialmente generate, contribuendo a migliorare sensibilmente la qualità di generazione.
Questa sinergia tra CNNs e GANs produce un robusto metodo di estrazione delle features e analisi delle immagini (CNN), migliorando le capacità del modello di generare maschere di immagine accurate e realistiche (GAN). Grazie a ciò, SAM è in grado di comprendere una vasta gamma di input grafici generando output ad alta precisione.

3.2 Grounded Segment Anything

Grounded Segment Anything combina le funzionalità di due modelli di machine learning appena descritti per creare un potente framework che risolva problemi complessi. Lo scopo è costruire un'infrastruttura articolata e modulare, dove gli sviluppatori abbiano la possibilità di modificare la combinazione di modelli per risolvere problemi sempre più specifici.

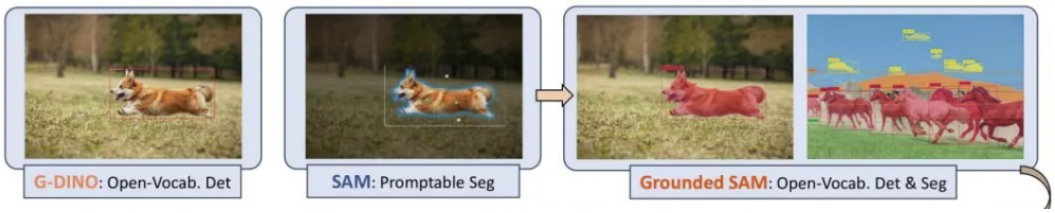

Figura 3.4. Grounded Segment Anything. Fonte: [6]

Questa interazione permette l'individuazione e la segmentazione di qualunque regione di immagine basandosi su un prompt testuale arbitrario, aprendo una porta sulla connessione di diversi modelli visuali. La pipeline di GSAM è estremamente versatile, in quanto si presta alla risoluzione di numerosissimi problemi complessi semplicemente incorporando on-top uno specifico modello.

3.3 Inizializzazione

All'interno di questa ricerca l'utilizzo di GSAM è stato circoscritto all'individuazione e alla segmentazione di oggetti dal dataset.
Si procede come di seguito: grounding Dino è stato utilizzato per tradurre specifici input testuali accuratamente validati in spunti grafici, predicendo determinate labels e producendo le relative bounding boxes; SAM ha poi sfruttato le bounding boxes

prodotte da GD per creare maschere di immagine dettagliate, dalle quali sono state poi estratte le relative features.
Di seguito uno snippet di codice rappresentante classi e funzioni principali.

Listing 3.1. bounding boxes, risultati di rilevamento e funzione di rilevamento-segmentazione

```
@dataclass
class BoundingBox:
    xmin: int
    ymin: int
    xmax: int
    ymax: int
    @property
    def xyxy(self) -> List[float]:
        return [self.xmin, self.ymin, self.xmax, self.ymax]
@dataclass
class DetectionResult:
    score: float
    label: str
    box: BoundingBox
    mask: Optional[np.array] = None
    @classmethod
    def from_dict(cls, detection_dict: Dict) -> 'DetectionResult':
        return cls(score=detection_dict['score'],
                   label=detection_dict['label'],
                   box=BoundingBox(xmin=detection_dict['box']['xmin'
                       ],.....))

def grounded_segmentation(
    image: Union[Image.Image, str],
    labels: List[str],
    threshold: float = 0.3,
    polygon_refinement: bool = False,
    detector_id: Optional[str] = None,
    segmenter_id: Optional[str] = None
) -> Tuple[np.ndarray, List[DetectionResult]]:
    if isinstance(image, str):
        image = load_image(image)
    detections = detect(image, labels, threshold, detector_id)
    detections = segment(image, detections, polygon_refinement,
        segmenter_id)
    return np.array(image), detections
```

Capitolo 4

Object recognition e segmentation

4.1 Descrizione dell'idea progettuale

L'idea di progetto consiste nell'analizzare separatamente le immagini sane e quelle anomale. Per ogni tipologia di immagini è stato utilizzato GDINO per individuare determinate classi di oggetti da specifici prompt testuali (opportunamente validati per massimizzare l'output) e poi SAM per segmentare ogni bounding box individuata da GDINO e generare le maschere d'immagine. Successivamente entrambi gli insiemi di maschere sono stati ridimensionati e normalizzati secondo media e deviazione standard di ogni maschera, per rendere i dati facilmente processabili dalla CNN per l'estrazione delle features (Capitolo 5).

4.2 Segmentazione delle anomalie

Si estrae dal dataset ciò che l'algoritmo individua essere una maschera non buona, anomala. Per fare ciò è necessario validare il modello con differenti combinazioni di prompt, per poi scegliere l'unione di input che produce il maggior numero di maschere con la migliore accuratezza. Di seguito lo snippet di codice.

Listing 4.1. codice utilizzato per la validazione

```
max_masks = -1
best_prompts = None
best_detections = None
best_scores_sum = 0
cont = 0
temp = 0
for prompts in unique_combinations:
```

```
    print("combo n.", cont)
    print("best prompts so far:",best_prompts)
    num = 0
    scores_sum = 0
    for i in range(len(bad_val_set)):
        try:
            image_array, detections = grounded_segmentation(
                image=dataset_BAD.__getitem__(i)[0],
                labels=prompts,
                threshold=threshold,
                polygon_refinement=True,
                detector_id=detector_id,
                segmenter_id=segmenter_id
            )
            num += len(detections)  # Sum of all masks found with
                this prompt
            scores_sum += sum([detection.score for detection in
                detections])  # Sum of all accuracy scores for
                subsequent comparison
            print("\timmagine n.", i, "/", len(bad_val_set)-1)
        except IndexError:
            print(f"\tIndexError at image index {i}. Skipping this
                image.")
            continue
    # Change prompt
    if num > max_masks:
        max_masks = num
        best_prompts = prompts
        best_scores_sum = scores_sum
        # best_detections = detections
    elif num == max_masks:
        if scores_sum > best_scores_sum:
            best_prompts = prompts
            best_scores_sum = scores_sum
            # best_detections = detections
    cont += 1
print("so the best prompts are: ",best_prompts)
```

Una volta ottenuta la combinazione di prompt che massimizza l'object detection, si procede applicandola a tutte le immagini dell'uva malata all'interno del train, validation e test set, ottenendo per ogni immagine diverse bounding box rappresentanti le anomalie e le corrispondenti maschere.

4.2.1 Esempio illustrativo

Di seguito un esempio illustrativo.

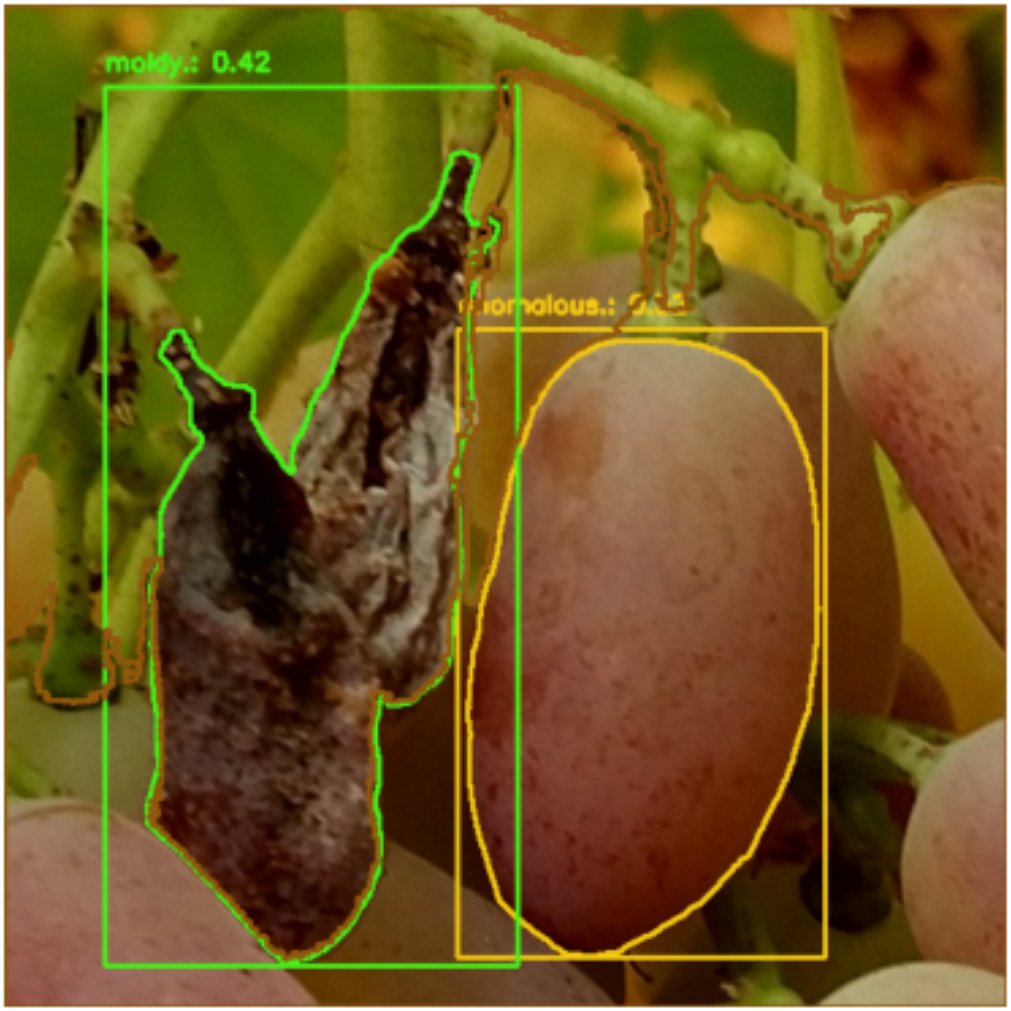

Figura 4.1. anomalie riconosciute in un campione di uva malata

.

si nota come l'algoritmo abbia riconosciuto con successo le regioni anomale dell'immagine con le loro rispettive labels, identificandone forme e perimetri.

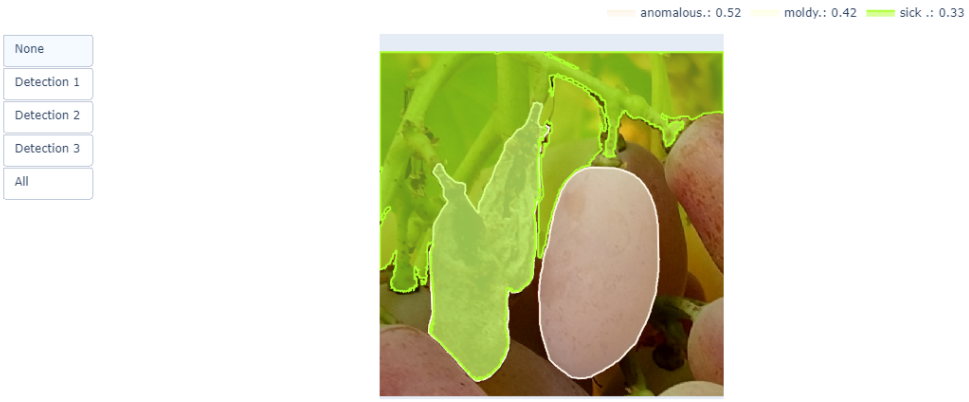

Figura 4.2. un grafico delle diverse anomalie divise per label

Sono state ottenute delle bounding boxes da dare in pasto alla funzione di segmentazione per ottenere le maschere, che si presentano come di seguito.

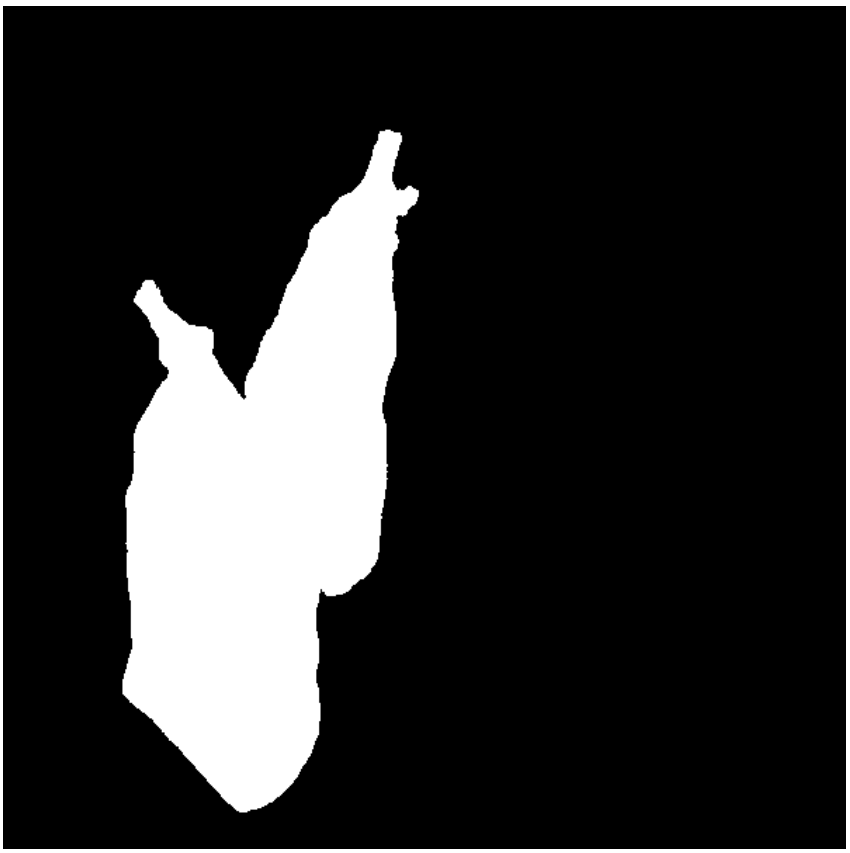

Figura 4.3. un campione di maschera anomala estratta

4.2.2 Segmentazione e normalizzazione

Le maschere ottenute da ogni insieme di immagini vengono ora ridimensionate e normalizzate. Viene calcolata la media e la deviazione standard di tutti i pixel di tutte le immagini ridimensionate per poi normalizzarle secondo queste ultime:

- **Media** (μ):

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

  Dove x_i rappresenta il valore di un singolo pixel e N è il numero totale di pixel.

- **Deviazione standard** (σ):

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}$$

  Dove μ è la media dei pixel e x_i è il valore del singolo pixel.

Normalizzazione di un singolo pixel:

$$x_i' = \frac{x_i - \mu}{\sigma}$$

Di seguito lo snippet di codice.

Listing 4.2. funzioni di normalizzazione

```
def resize_and_normalize_masks(image_list, desired_size=(244, 244)):
    """
    Resizes and normalizes a list of images.
    Parameters:
    - image_list: list of numpy arrays representing the images
    - desired_size: tuple indicating the desired size for resizing (
        default is (244, 244))
    Returns:
    - n_bad_masks: list of resized and normalized images
    """
    # Step 1: Resize each array
    resized_arrays = [cv2.resize(arr, desired_size) for arr in
        image_list]
    # Compute mean and standard deviation
    mean_value = np.mean(np.stack(resized_arrays))
    std_value = np.std(np.stack(resized_arrays))
    # Step 2: Normalize each array
```

```
    n_bad_masks = [(arr - mean_value) / std_value for arr in
        resized_arrays]
    return n_bad_masks
# Example usage:
# resized_and_normalized_images = resize_and_normalize_images(
    bad_masks)
```

4.3 Segmentazione dell'uva sana

Per quanto riguarda invece il processamento dell'uva sana, si prosegue in modo leggermente differente. Dal momento che, per scelte di progetto, si considera ora solo la parte sana del dataset, non vi è bisogno di procedere con ulteriori validazioni di prompts testuali. Sulla base di alcuni test e del paper del modello, si nota come le performance su input più facili e generici (come possono essere quelli utilizzati per cercare una situazione standard e non anomala) sono migliori e molto simili quanto ad accuracy. Si procede quindi iterando su una combinazione di prompts generici per segmentare le maschere positive.

4.3.1 Esempio illustrativo

Di seguito un esempio illustrativo.

Figura 4.4. uva sana riconosciuta

Si nota come l'algoritmo abbia riconosciuto con successo le regioni sane dell'immagine con le loro rispettive labels, identificandone forme e perimetri.

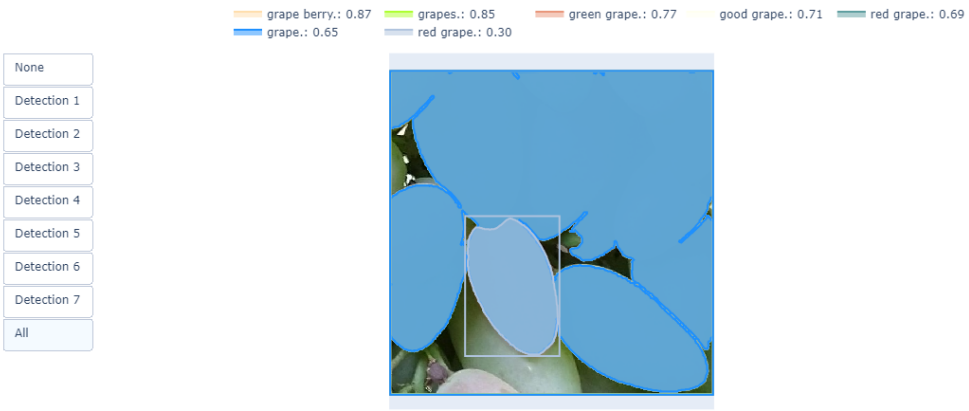

Figura 4.5. un grafico delle bounding boxes sane divise per label

Sono state ottenute ora delle bounding boxes da dare in pasto, come precedentemente, alla stessa funzione di segmentazione per ottenere le maschere, che si presentano come di seguito.

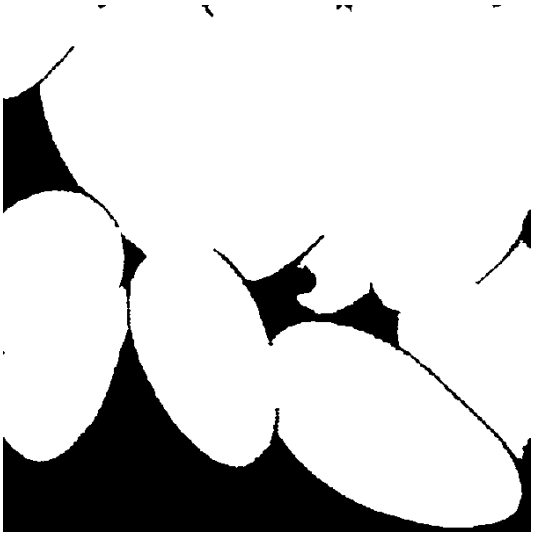

Figura 4.6. un campione di maschera sana estratta

4.3.2 Segmentazione e normalizzazione

Le maschere ottenute da ogni insieme di immagini vengono, come fatto prima, ridimensionate e normalizzate utilizzando i medesimi criteri.

Capitolo 5

Estrazione delle features dalle maschere d'immagine

5.1 Convolutional Neural Network

In questo capitolo viene affrontata l'estrazione delle features dalle maschere di immagine segmentate di cui al capitolo 4, ad opera di una rete neurale convoluzionale (CNN).
Una CNN è un algoritmo di deep learning che prende in input un'immagine, assegna una determinata importanza (pesi) a vari aspetti e oggetti della stessa ed è in grado di differenziarli gli uni dagli altri. Il pre-processamento delle immagini richiesto in input da una CNN è di gran lunga più esiguo rispetto a quello richiesto da altri modelli di classificazione. I modelli classici richiedono un'estrazione manuale delle features attraverso l'applicazione di vari kernel sull'immagine; una CNN ben allenata ha la capacità di apprendere automaticamente gli stessi filtri e utilizzarli per individuare features dalle immagini in autonomia.

L'architettura di una rete neurale convoluzionale è analoga alla struttura connettiva dei neuroni di un cervello umano ed è ispirata all'organizzazione della corteccia visiva (corteccia a tipo sensoriale) dello stesso.
Singoli neuroni rispondono agli stimoli solo in una ristretta regione del campo visivo, il campo recettivo. Una raccolta di tali campi, sovrapponendosi, copre l'intera area visiva.
Una CNN è in grado di catturare con straordinaria precisione le dipendenze spaziali e temporali di un'immagine. La sua architettura permette alla rete di adattarsi meglio alla complessità dell'input, grazie alla riduzione dei parametri coinvolti e alla riusabilità dei pesi.

In generale l'architettura di una CNN [15] è sviluppata su tre layer: un layer convoluzionale, un layer di pooling e un layer totalmente connesso.

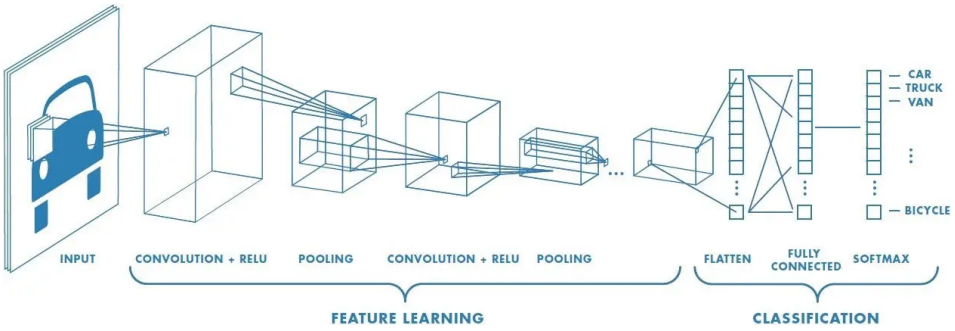

Figura 5.1. architettura di una CNN. Fonte: [7]

5.1.1 Layer convoluzionale

Il layer convoluzionale rappresenta il nucleo della CNN e si fa carico della maggior parte del carico computazionale. Questo strato necessita dei dati di input, un kernel e una mappa delle features e esegue un prodotto scalare tra due matrici, il kernel e la porzione di campo recettivo. Il kernel ha dimensione minore rispetto all'immagine ma è più "profondo". Ergo se l'immagine fosse ad esempio composta da 3 canali RGB, il kernel (HxW) avrebbe dimensioni spaziali ridotte ma sarebbe profondo abbastanza da coprire tutti i canali.

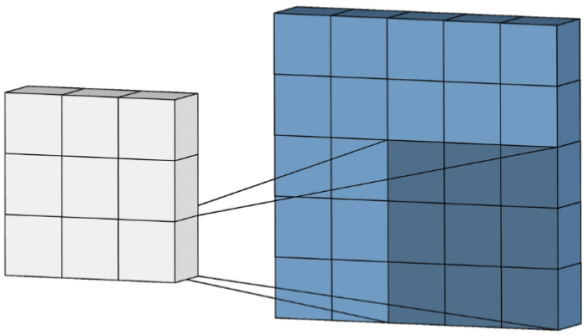

Figura 5.2. esempio illustrativo della convoluzione. Fonte: [8]

Durante il passaggio in avanti (feed-forward della rete), il kernel scorre tutta l'immagine lungo l'altezza e la larghezza cercando features, producendo la rappresentazione di quella regione ricettiva. L'unione dei prodotti scalari su tutta l'immagine produce una rappresentazione bidimensionale dell'immagine, nota come mappa di attivazione, che rappresenta il risultato dell'applicazione del kernel in ciascuna posizione spaziale

dell'immagine. La dimensione di scorrimento del kernel è chiamata passo.
Supponendo di avere ad esempio un'immagine di dimensione $WxWxD$ e un numero di kernels $Dout$ con una dimensione spaziale F, passo S e padding P, si può calcolare la dimensione dell'output come segue:

$$Wout = \frac{W - F + 2P}{S} + 1$$

Ciò, su 3 canali, produrrà una dimensione dell'output $WoutxWoutxWout$

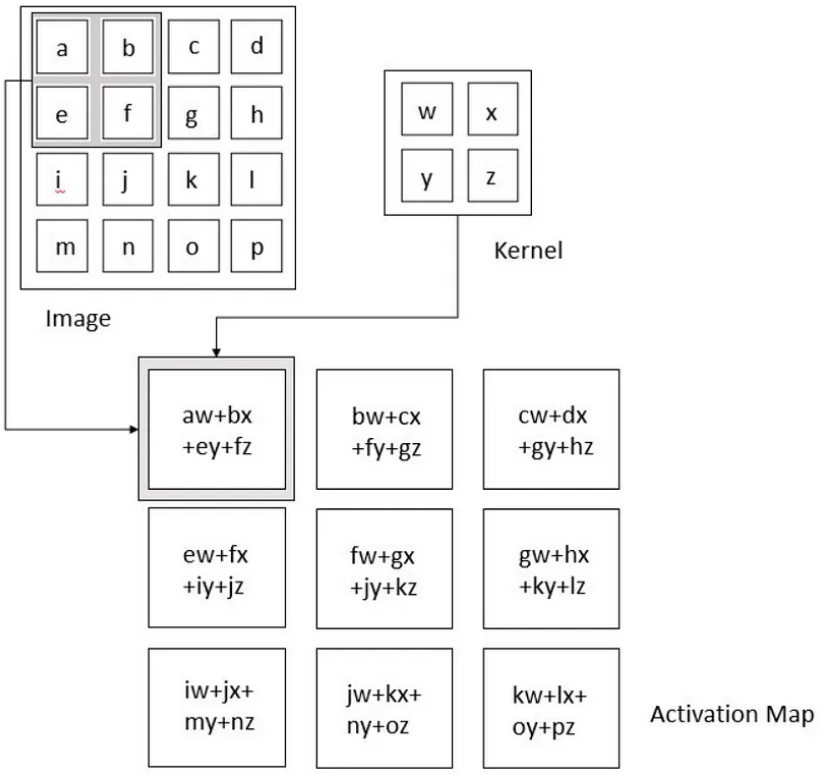

Figura 5.3. operazione di convoluzione. Fonte: [9]

La convoluzione fa leva su tre principi fondamentali: l'interazione sparsa, la condivisione di parametri e la rappresentazione equivariante. Passiamo ad analizzarli.
Gli strati di reti neurali più banali applicano moltiplicazioni matriciali con matrici di parametri, facendo interagire ogni elemento dell'input con ogni elemento dell'output. Le dimensioni ridotte dei kernels delle CNN permettono interazioni sparse, rendendo possibile l'estrazione di features grandi centinaia o migliaia di pixels da immagini che potrebbero contarne milioni. Ciò significa che non c'è necessità di memorizzare tutti i parametri, riducendo la quantità di memoria necessaria e migliorando l'efficienza statistica del modello.

Figura 5.4. esempio di scomposizione in features. Fonte: [10]

Se calcolare una determinata features in un punto spaziale (x1, y1) può risultare utile, dovrebbe risultarlo anche calcolarla in un altro punto (x2, y2). Ciò vuol dire che per una rappresentazione bidimensionale, come una mappa di attivazione, i neuroni riutilizzano lo stesso insieme di pesi, aggiornandone i valori tramite backpropagation e gradient descend. In una rete tradizionale i pesi non vengono rivisitati, mentre la condivisione dei parametri delle CNN consente di ottenere un output da un determinato input utilizzando gli stessi pesi applicati altrove.
La compartecipazione dei parametri rende anche la rete equivariante alla traslazione: se cambia un input, l'output cambierà con le stesse modalità.

Non linearità

La convoluzione rappresenta un'operazione lineare su immagini, le quali però nella realtà sono molto lontane dalla linearità. Spesso, appena dopo il layer convoluzionale, vengono introdotti layer non lineari per includere la non linearità nelle mappe di attivazione.
Ci sono vari tipi di funzioni non lineari che posso essere generalmente applicate in questi strati, le più comuni sono le seguenti:

- Sigmoide: è una funzione non lineare che si presenta nella seguente forma.

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$

  Normalizza i valori, mantenendo i segnali all'interno di un intervallo limitato. La funzione sigmoidea soddisfa questa proprietà:

$$\frac{d}{dt}\sigma(t) = \sigma(t)(1 - \sigma(t))$$

  particolarmente utile per l'allenamento delle reti neurali, poiché il gradiente della sigmoide può essere espresso in termini della funzione sigmoide stessa, rendendo la derivata facile da calcolare durante la fase di retropropagazione del gradiente (backpropagation).

Per valori molto positivi o negativi di t, però, la derivata diventa molto piccola. Ad esempio, per t molto positivo, $\sigma(t)$ si avvicina ad 1 e quindi $\sigma(t)(1-\sigma(t))$ è prossimo allo 0 (e viceversa per $\sigma(t)$=0). Questo fenomeno è noto come vanishing gradient problem, cioè il gradiente diventa così piccolo da non permettere un aggiornamento efficace dei pesi durante l'allenamento della rete neurale.
Ergo, se un neurone riceve un segnale molto positivo o molto negativo, la funzione sigmoide restituisce un output molto vicino a 1 o a 0. Questo può portare a un aggiornamento lento o nullo dei pesi poiché il gradiente è quasi nullo in queste regioni, e quindi non si hanno modifiche significative nei pesi durante l'apprendimento.

- ReLU: sta per Rectified Linear Unit, una funzione definita come la parte positiva del suo argomento, limitata a zero (annoverata tra le funzioni rampa, è l'analogo di un raddrizzatore in elettronica.).

$$f(k) = max(0, k)$$

Trasforma l'output utilizzando il seguente algoritmo: se l'input è minore di 0, restituisce 0; se l'input è maggiore o uguale a 0, restituisce l'input. La funzione ReLU è più affidabile, accelera la convergenza di circa 6 volte ed è decisamente meno suscettibile al problema della scomparsa del gradiente. Sfortunatamente, la ReLU può risultare fragile durante l'allenamento. Un gradiente di grandi dimensioni che la attraversa potrebbe aggiornarla in modo da non aggiornare ulteriormente il neurone. Ciò può essere tuttavia risolto validando il learning rate.

5.1.2 Layer di pooling

Il layer di pooling, o strato di sottocampionamento, si occupa della riduzione dimensionale dell'input. Come nella convoluzione, il pooling fa scorrere su tutta l'immagine un filtro che però non possiede pesi. Questo kernel applica invece una funzione di aggregazione agli elementi nel campo percettivo, popolando un'array di output di dimensioni minori che predilige le features dominanti e riduce il rumore. Esistono varie operazioni di pooling (rectangular neighborhood, L2 norm of the rectangular neighborhood, max pooling, average pooling), quella però generalmente più applicata è il max pooling, che riporta il massimo valore di ogni insieme.

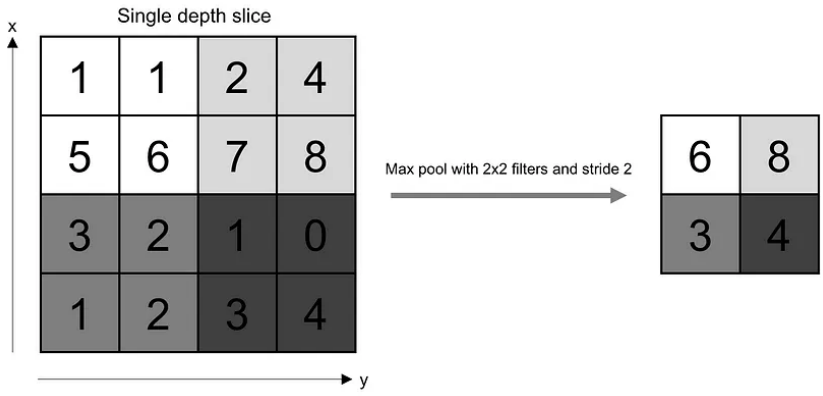

Figura 5.5. esempio di max pooling. Fonte: [11]

Se avessimo ad esempio una mappa di attivazione di dimensione $WxWxD$, un kernel di pooling di dimensione F e passo S, allora la dimensione dell'output sarebbe determinata dalla seguente formula:

$$Wout = \frac{W - S}{F} + 1$$

In ogni caso il pooling conserva l'invarianza alla traslazione, rendendo riconoscibile un oggetto indipendentemente dalla posizione all'interno del frame.
Il layer convoluzionale e il layer di pooling costituiscono l'i-esimo strato di una CNN. Il numero di questi strati può aumentare in relazione alla complessità delle immagini, incrementando la capacità di catturare dettagli più accurati, ma con un conseguente aumento del costo computazionale.
Finito il processo di pooling, il modello ha terminato la fase di apprendimento ed è ora in grado di riconoscere le features dalle immagini.

5.1.3 Layer completamente connesso - classificazione

I pixel dell'immagine in input non sono direttamente connessi con lo strato di output. Nel layer completamente connesso, ogni nodo dello strato di output è connesso direttamente a un nodo nel layer precedente. Questo layer esegue la classificazione basandosi sulle features estratte in precedenza e sui differenti filtri. Sfruttando una funzione softmax, dopo una serie di epoche, distingue tra features dominanti e di basso livello per poi classificarle producendo una probabilità compresa tra 0 e 1.

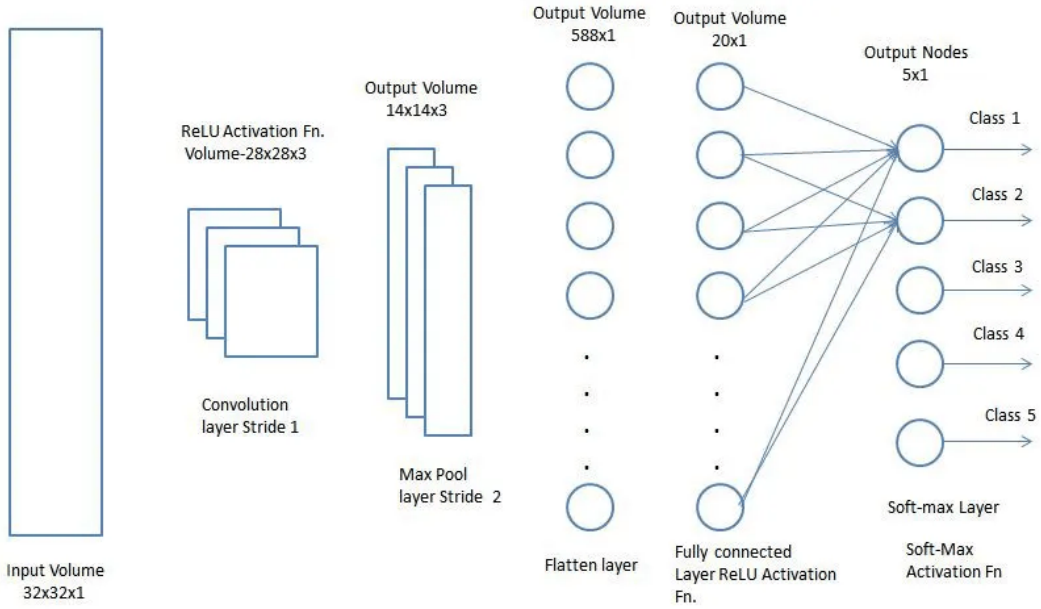

Figura 5.6. struttura completa. Fonte: [12]

In matematica, una funzione softmax, o funzione esponenziale normalizzata, è una generalizzazione di una funzione logistica che mappa un vettore K-dimensionale z di valori reali arbitrari in un vettore K-dimensionale $\sigma(z)$ di valori compresi in un intervallo (0,1) la cui somma è 1. la funzione è la seguente:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

In sintesi, il layer completamente connesso mappa la rappresentazione tra l'input e l'output.

5.2 VGG16

La rete utilizzata nella presente ricerca è VGG16 (Visuale Geometry Group 16). VGG16 è una CNN considerata tra i migliori modelli di computer vision al momento. Presentata alla ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) del 2014 da Karen Simonyan e Andrew Zisserman del Visual Geometry Group, Dipartimento di Engineering Science dell'Università di Oxford, la rete è descritta nel paper intitolato "Very Deep Convolutional Networks for Large-Scale Image Recognition".
I creatori del modello hanno aumentato la profondità utilizzando un'architettura con filtri di convoluzione di dimensioni ridotte (3 x 3), mostrando un miglioramento rispetto a configurazioni precedenti. La rete ha dai 16 ai 19 layers, ammontando

circa a 138 parametri utili all'allenamento. L'algoritmo di classificazione di VGG16 è in grado di classificare 1000 immagini di 1000 categorie differenti con un'accurztezza del 92.7%

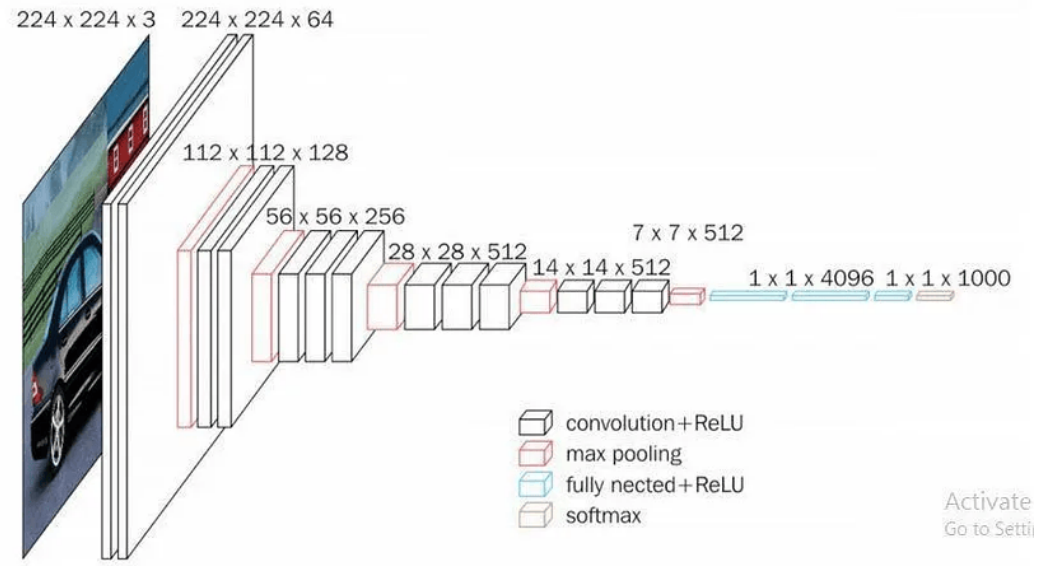

Figura 5.7. architettura della VGG16. Fonte: [13]

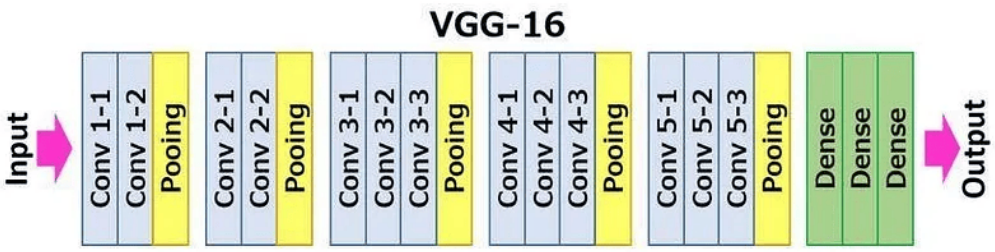

Figura 5.8. layers della VGG16. Fonte: [14]

La rete prende in input un tensore 224 X 224 con 3 canali RGB e conta 13 layers convoluzionali, 5 layer di max pooling e 3 strati densi. Solo 16 di questi sono però pesati e quindi contribuiscono all'addestramento dei parametri. L'aspetto più distintivo di VGG16 è che, invece di utilizzare un gran numero di iperparametri, si concentra sull'impiego di livelli di convoluzione con filtri di dimensioni 3x3 e passo 1, utilizzando sempre lo stesso padding e uno strato di max-pooling con filtri 2x2 e passo 2. I livelli di convoluzione e max-pooling sono disposti in modo coerente lungo tutta l'architettura. Il primo strato di convoluzione (Conv-1) utilizza 64 filtri, Conv-2 ne ha 128, Conv-3 ne impiega 256, mentre Conv-4 e Conv-5 utilizzano ciascuno 512 filtri.
Dopo la sequenza di livelli di convoluzione, si trovano tre livelli completamente

connessi: i primi due hanno ciascuno 4096 canali, mentre il terzo strato esegue la classificazione ILSVRC a 1000 classi, contenendo quindi 1000 canali (uno per ciascuna classe). L'ultimo strato è uno strato softmax.

5.3 Estrazione delle features

Si procede ora con l'applicazione della CNN alle maschere segmentate di cui al capitolo 4, per estrarre le features. In questo lavoro è stata utilizzata l'architettura VGG16 pre-addestrata (pesata su Imagenet), escludendo lo strato superiore (top layer), responsabile della classificazione finale. In tal modo, l'output del modello è costituito unicamente dalle features estratte dai livelli convolutivi e di pooling. Queste caratteristiche sono state successivamente fornite in input a un'altra rete neurale, appositamente addestrata per eseguire il compito di classificazione specifico. Questo approccio consente di sfruttare la capacità di VGG16 di estrarre features complesse dalle immagini, mantenendo la flessibilità di un classificatore personalizzato. Le immagini sono state pre-processate combinando TensorFlow e PyTorch per poi procedere con l'effettiva estrazione delle features con VGG16 in TensorFlow. Di seguito vengono indicate le fasi principali del processamento immagine eseguito:

1. Le maschere di immagine sono in bianco e nero, dispongono quindi di un solo canale RGB. É necessario creare un'immagine a 3 canali per renderla coerente con l'input di VGG. Questo viene fatto creando tre copie identiche di ogni immagine, "impilando" i canali lungo l'asse del colore per convertire il canale singolo in immagini a 3 canali.

2. Ogni immagine RGB viene quindi convertita in un tensore PyTorch. Successivamente tutte le immagini (divise per train, validation e test set) vengono organizzate in un unico tensore tridimensionale, creando una struttura dati che può gestire più immagini contemporaneamente

3. Questo tensore viene convertito in un array NumPy.
   TensorFlow, che gestisce VGG, opera principalmente con array NumPy e non supporta direttamente alcune operazioni necessarie per il processamento del modello.

4. Le immagini vengono pre-processate nuovamente con un modulo di VGG, per adattarle definitivamente all'input normalizzando i pixel e ridimensionandole.

5. Le immagini pre-processate vengono passate attraverso VGG per estrarre le features. L'output del modello, con il livello "block5pool" selezionato come output finale, produrrà features di dimensioni (7, 7, 512) rappresentando le mappe di attivazione dell'ultimo strato.

Di seguito uno snippet di codice.

Listing 5.1. processamento delle immagini

```
# Step 1: Create 3-channel images
list_of_rgb_arrays = [np.stack((img, img, img), axis=-1) for img in
    n_good_masks_train]
# Step 2: Convert to PyTorch tensors
list_of_tensors = [torch.tensor(arr) for arr in list_of_rgb_arrays]
# Step 3: Organize into a single tensor
final_tensor = torch.stack(list_of_tensors)
# Print shapes for verification
#print(f"Shape of final tensor: {final_tensor.shape}")
final_tensor_np = final_tensor.numpy()##i convert to numpy cause
    tensor does not support a negative strip operation that the model
    does!!
images = preprocess_input(final_tensor_np)  # Preprocess input for
    VGG16
# Extract features
good_masks_train_features = model.predict(images)
# The shape of features will be (number of images, 7, 7, 512) for
    VGG16 with 'block5_pool' layer
```

In sintesi: Il codice prende un insieme di maschere a canale singolo, le trasforma in immagini RGB, le converte in tensori PyTorch, le riporta a NumPy, le preprocessa per l'input al modello VGG16 ed estrae le features convoluzionali dallo strato 'block5pool' di VGG16, che possono poi essere usate per la classificazione finale.

5.4 Ribilanciamento del dataset

Una volta eseguita l'estrazione delle features da entrambe le classi di immagini (sane e anomale) viene riconsiderato il bilanciamento del nuovo dataset, input della classificazione. Le immagini di uva sana hanno prodotto un output caratteristicamente molto vario, generando un numero di features di molto maggiore a quello dell'uva anomala. Ciò potrebbe compromettere l'efficacia del modello nel riconoscere correttamente le anomalie.
Per mitigare questo problema, è stato applicato un downsampling sulle features estratte dalle immagini di uva sana, riducendo il numero di campioni in modo da bilanciare le due classi. Questo intervento è finalizzato a garantire una distribuzione più equilibrata nel dataset e a migliorare le performance del modello durante la fase di classificazione nonchè a ridurne la complessità computazionale.

Capitolo 6

Classificazione - Anomaly detection

Si procede ora con la fase finale del progetto, la classificazione.
L'operazione è svolta da una semplice rete neurale a 4 layers, che opera su un nuovo dataset temporaneo costituito dalle features estratte di cui al capitolo 5

6.1 Dataset delle features

Un nuovo dataset è stato creato con le features estratte dalle immagini. Esse vengono convertite in tensori PyTorch, divisi in addestramento, validazione e test. Successivamente vengono assegnate le etichette binarie alle maschere, dove le immagini delle maschere "buone" sono etichettate con il valore 0, mentre le maschere "cattive" sono etichettate con il valore 1. Queste etichette vengono generate per ciascuno dei tre set di dati.
Dopo aver assegnato le etichette, le features e le etichette delle maschere "buone" e "cattive" vengono concatenate, formando un unico insieme di dati per ogni set (addestramento, validazione e test). Questo insieme viene quindi organizzato in un dataset strutturato, chiamato TensorDataset, che associa le features con le rispettive etichette.
Infine, si creano i dataloaders, che suddividono i dati in batch della dimensione specificata (in questo caso, 32), facilitando l'elaborazione efficiente durante l'addestramento del modello. I dataloaders consentono al modello di accedere ai dati in maniera strutturata, permettendo di caricare i dati in memoria in modo ottimale durante l'addestramento, la validazione e il test.
Di seguito lo snippet di codice rappresentante le operazioni di cui sopra:

Listing 6.1. processamento maschere e creazione del dataset

```
good_masks_train_features = torch.tensor(
    good_features_train_downsampled, dtype=torch.float32)
good_masks_val_features = torch.tensor(good_features_val_downsampled,
     dtype=torch.float32)
good_masks_test_features = torch.tensor(
    good_features_test_downsampled, dtype=torch.float32)
bad_masks_train_features = torch.tensor(bad_masks_train_features,
    dtype=torch.float32)
bad_masks_val_features = torch.tensor(bad_masks_val_features, dtype=
    torch.float32)
bad_masks_test_features = torch.tensor(bad_masks_test_features, dtype
    =torch.float32)
good_labels_train = torch.zeros(good_masks_train_features.size(0))
bad_labels_train = torch.ones(bad_masks_train_features.size(0))
good_labels_val = torch.zeros(good_masks_val_features.size(0))
bad_labels_val = torch.ones(bad_masks_val_features.size(0))
good_labels_test = torch.zeros(good_masks_test_features.size(0))
bad_labels_test = torch.ones(bad_masks_test_features.size(0))
features_train = torch.cat((good_masks_train_features,
    bad_masks_train_features), dim=0)
features_val = torch.cat((good_masks_val_features,
    bad_masks_val_features), dim=0)
features_test = torch.cat((good_masks_test_features,
    bad_masks_test_features), dim=0)
train_labels = torch.cat((good_labels_train, bad_labels_train), dim
    =0)
val_labels = torch.cat((good_labels_val, bad_labels_val), dim=0)
test_labels = torch.cat((good_labels_test, bad_labels_test), dim=0)
# Create a dataset
train_dataset = TensorDataset(features_train, train_labels)
val_dataset = TensorDataset(features_val, val_labels)
test_dataset = TensorDataset(features_test, test_labels)
batch_size = 32
train_loader, val_loader, test_loader = create_dataloaders(
    train_dataset, val_dataset, test_dataset, batch_size)
```

6.2 Rete neurale utilizzata

Per la fase finale di classificazione dei dati è stata sviluppata e implementata appositamente una rete neurale artificiale, denominata *Simple Classifier*. Questa rete è stata progettata come una feedforward neural network composta da tre strati pienamente connessi (fully connected), così da rimpiazzare l'ultimo strato fully connected rimosso dal modello VGG durante la fase di estrazione delle features. Ogni strato è seguito da una funzione di attivazione per introdurre la necessaria non

linearità, consentendo al modello di apprendere rappresentazioni complesse dai dati. Di seguito lo snippet di codice:

Listing 6.2. Simple classifier

```
class SimpleClassifier(nn.Module):    # I build a simple feedforward
    NN classifier
    def __init__(self):
        super(SimpleClassifier, self).__init__()
        self.fc1 = nn.Linear(7 * 7 * 512, 512)  #fully connected
            layer that takes an input of size 7 * 7 * 512 and outputs
            512 features.
        self.fc2 = nn.Linear(512, 128)          #fully connected
            layer that takes 512 features as input and outputs 128
            features.
        self.fc3 = nn.Linear(128, 1)            #fully connected
            layer that takes 128 features and outputs a single value,
            used for binary classification.
    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flattens the input tensor except
             for the batch dimension. If x is of shape (batch_size, 7,
             7, 512), it will be reshaped to (batch_size, 7 * 7 * 512)
            .
        x = F.relu(self.fc1(x))    # relu activation to the first
            layer
        x = F.relu(self.fc2(x))    # relu activation to the second
            layer
        x = torch.sigmoid(self.fc3(x))  #  sigmoid activation
            function to the output of the final layer, producing a
            value between 0 and 1, for binary classification
        return x
model = SimpleClassifier()
```

6.2.1 Primo strato - fc1

Lo strato "fc1" è un livello fully connected che riceve in input un tensore di dimensione 7 x 7 x 512, derivato dall'uscita dell'ultimo blocco del modello VGG16 utilizzato, e lo mappa in uno spazio di dimensione 512. Questo strato riduce drasticamente la dimensionalità del tensore, riducendolo a 512 elementi, pur mantenendo una rappresentazione ricca delle features. La dimensione iniziale (7 x 7 x 512) rappresenta un'estrazione dettagliata di features spaziali e canali, mentre la riduzione della dimensionalità a 512 rende il modello più efficiente, facilitando un processo decisionale su un numero più gestibile di features senza perdere informazioni essenziali.

6.2.2 Secondo strato - fc2

Lo strato "fc2" è un altro livello fully connected che riceve in input i 512 elementi dallo strato precedente e li riduce a 128. Questo livello consente una successiva riduzione della complessità, permettendo al modello di concentrare l'attenzione su features più raffinate e compatte. Ridurre ulteriormente la dimensionalità aiuta a comprimere l'informazione in un formato che faciliti la classificazione. I 128 elementi sono sufficientemente piccoli per contenere rappresentazioni utili, ma abbastanza grandi da evitare la perdita eccessiva di informazioni.

6.2.3 Terzo strato - fc3

Lo strato finale "fc3" è un livello fully connected che riduce i 128 elementi a un singolo valore scalare. Questa quantità rappresenta un valore continuo compreso tra 0 e 1, rappresentante la probabilità che l'input appartenga a una specifica classe.

6.2.4 Funzioni di attivazione

Per garantire la capacità della rete di apprendere funzioni non lineari, ogni strato fully connected è seguito da una funzione di attivazione. In particolare:

1. La funzione di attivazione ReLU è applicata dopo il primo e il secondo strato. Come visto nel capitolo precedente, la funzione introduce una non linearità, sostituendo tutti i valori negativi con zero, mantenendo intatti i valori positivi. Questa scelta è giustificata dalla sua semplicità ed efficienza computazionale..

$$\text{ReLU}(x) = \max(0, x)$$

2. L'output dell'ultimo strato è passato attraverso la funzione sigmoide, che mappa il valore scalare risultante in un intervallo compreso tra 0 e 1.

$$\text{Sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

   Come spiegato nella sottosezione non linearità del capitolo precedente, la funzione sigmoide è adatta per compiti di classificazione binaria poiché trasforma un valore continuo in una probabilità, che può essere interpretata come la probabilità che l'input appartenga a una determinata classe.

6.2.5 Operazioni svolte dal modello

Nel metodo "forward", il modello elabora il tensore di input in diverse fasi:

1. Flattening: la prima operazione è il "flattening" del tensore di input, ovvero la trasformazione del tensore multidimensionale (in questo caso con dimensione (7, 7, 512) in un vettore unidimensionale di lunghezza 7 x 7 x 512 = 25088. Questo è necessario poiché i livelli fully connected operano su vettori e non su tensori multidimensionali.

2. Propagazione nei Fully Connected Layers: dopo il flattening, l'input passa attraverso i tre livelli fully connected, ciascuno con una riduzione dimensionale e una funzione di attivazione non lineare:

   - Primo strato: l'input viene ridotto da 25088 a 512 → ReLU.
   - Secondo strato: riduzione da 512 a 128 → ReLU.
   - Terzo strato: riduzione a 1 singolo valore scalare.

3. Output Finale: l'output viene infine passato attraverso la funzione sigmoide, che trasforma il valore in una probabilità compresa tra 0 e 1, rappresentante l'appartenenza a una delle due classi.

6.2.6 Motivazioni e giustificazioni delle scelte di progetto

Dimensioni degli Strati

La scelta delle dimensioni (512, 128, 1) per gli strati fully connected è dettata dall'esigenza di ridurre progressivamente la complessità delle features estratte, mantenendo comunque un livello di informazione sufficiente per il compito di classificazione. Una riduzione troppo brusca potrebbe portare a una perdita di informazioni, mentre una troppo lenta risulterebbe inefficiente.

Attivazioni ReLU e Sigmoid

La funzione di attivazione ReLU è utilizzata per le sue proprietà di efficienza e prevenzione di problemi come il vanishing gradient. La funzione sigmoid all'output è invece cruciale per l'interpretazione probabilistica dei risultati.

6.3 Parametri e metriche utilizzati

6.3.1 Loss Function

Come funzione di perdita (loss function) è stata utilizzata la Binary Cross Entropy Loss (BCELoss), una scelta comune per compiti di classificazione binaria in cui l'output del modello è un valore di probabilità compreso tra 0 e 1. Per un

singolo campione, la BCELoss è definita come segue:

$$\text{BCELoss}(x, y) = -[y \cdot \log(x) + (1 - y) \cdot \log(1 - x)]$$

dove:

- x rappresenta la probabilità predetta dal modello (un valore scalare compreso tra 0 e 1),
- y è l'etichetta target (0 o 1, che indica la classe vera).

In questo modello x è l'output passato attraverso una funzione sigmoide, che trasforma il risultato in una probabilità compresa tra 0 e 1.
La BCELoss penalizza il modello in base alla differenza tra la probabilità predetta e l'etichetta reale (label). Il suo scopo è quello di incoraggiare il modello a produrre probabilità vicine a 1 per esempi positivi (in questo caso, le anomalie sono indicate con l'etichetta 1) e vicine a 0 per esempi negativi (ovvero le immagini di uva sana, che hanno l'etichetta 0).
L'uso della BCELoss è particolarmente adatto in questo contesto perché si sta affrontando un compito di classificazione binaria, in cui l'obiettivo è separare due classi distinte: uva sana e uva con anomalie. Inoltre, l'uso della funzione sigmoide sull'output permette alla BCELoss di operare direttamente su valori probabilistici, sfruttando direttamente il valore ideale (la probabilità) per ottimizzare la rete neurale e consentirle di distinguere efficacemente tra le due classi.

6.3.2 Ottimizzatore

L'ottimizzatore utilizzato in fase di train per il modello è "Adam".
Adam (Adaptive Moment Estimation) è un algoritmo di ottimizzazione molto popolare e potente. È progettato per migliorare l'efficienza e la performance rispetto ad altri algoritmi, come il Gradient Descent Stocastico (SGD) con o senza momentum.

Funzionamento di Adam

Adam combina le idee di due altri algoritmi di ottimizzazione: Momentum e RMSprop.

- Momentum accelera il gradiente nella direzione dei minimi locali e aiuta a superare i minimi locali del gradiente. Mantiene una "velocità" dei gradienti, che si accumula nel tempo. La velocità aggiorna i pesi in base alla combinazione del gradiente corrente e della velocità passata.

- RMSprop (Root Mean Square Propagation) modifica il tasso di apprendimento per ogni parametro in base alla media mobile quadratica dei gradienti. Questo aiuta a mantenere il tasso di apprendimento in un intervallo utile e impedisce che diventi troppo grande o troppo piccolo.

Passaggi dell'Algoritmo

Adam combina queste due idee e funziona come segue:

1. Stima dei Momenti (Momentum e RMSprop):
   - Primi Momenti (M): calcola la media mobile dei gradienti, simile al Momentum.
   - Secondi Momenti (V): calcola la media mobile delle variazioni quadrate dei gradienti, simile a RMSprop.
2. Aggiornamento dei Parametri: Adam utilizza queste stime per aggiornare i pesi del modello in modo più informato e adattivo.

Di seguito la formula dettagliata dell'algoritmo:

1. Calcolo dei Gradienti: Per un dato passo t:

$$g_t = \nabla_\theta J(\theta_t)$$

   dove g_t è il gradiente della funzione di perdita rispetto ai pesi θ.

2. Aggiornamento dei Moment:
   - Primi Momenti (m):

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

   - Secondi Momenti (v):

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

   dove β_1 e β_2 sono i parametri di decadimento esponenziale (di solito $\beta_1 = 0.9$ e $\beta_2 = 0.999$).

3. Correzione dei Bias: Per correggere il bias dei momenti all'inizio dell'addestramento:

(a) Primi Momenti (m) Corretto:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

(b) Secondi Momenti (v) Corretto:

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

4. Aggiornamento dei Pesi:

$$\theta_{t+1} = \theta_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

dove:

- α è il tasso di apprendimento (ad esempio 0.001).
- ϵ è un piccolo valore (ad esempio 10^{-8}) per evitare divisioni per zero.

Vantaggi di Adam

Adam è un ottimizzatore molto efficace che presenta tre principali vantaggi. In primo luogo, è adattativo: adatta automaticamente i tassi di apprendimento per ogni parametro, il che può accelerare la convergenza e trovare buone soluzioni più rapidamente rispetto all'uso di un tasso di apprendimento fisso. In secondo luogo, è computazionalmente efficiente: utilizza solo la prima e la seconda stima dei momenti, rendendolo molto economico in termini di memoria e calcolo. Infine, include una correzione dei bias che migliora la qualità dell'ottimizzazione, specialmente nelle fasi iniziali dell'addestramento, quando le stime dei momenti potrebbero essere imprecise.

6.3.3 Metriche di valutazione della performance del modello

Per valutare le performance del modello, sono state utilizzate diverse metriche fondamentali sia durante l'addestramento che nella fase di validazione.
Di seguito, viene fornita una spiegazione dettagliata di ciascuna metrica utilizzata:

- Train Loss: rappresenta la funzione di perdita calcolata sui dati di addestramento ad ogni epoca. Il valore della loss indica quanto il modello sta imparando durante l'addestramento e viene minimizzato tramite l'ottimizzazione. Una diminuzione della train loss nel tempo è indicativa di un buon apprendimento.

- Validation Loss: simile alla train loss, ma calcolata sui dati di validazione. Viene utilizzata per monitorare le prestazioni del modello su dati non visti

durante l'addestramento, aiutando a rilevare fenomeni di overfitting. Un valore di validation loss che continua a diminuire o rimane stabile indica che il modello generalizza bene.

- Validation Accuracy: è la percentuale di campioni correttamente classificati sul set di validazione. L'accuracy valuta semplicemente il numero di predizioni corrette rispetto al totale dei campioni. Tuttavia, in problemi di classificazione sbilanciata, può non essere sufficiente a valutare correttamente il modello.

- Validation True Positives (TP): indica il numero di campioni positivi correttamente identificati come tali dal modello. Nel contesto del problema, rappresentano le anomalie correttamente classificate come tali.

- Validation False Positives (FP): indica il numero di campioni negativi (sani) che sono stati erroneamente classificati come positivi (anomalie). Un alto valore di FP implica che il modello ha una bassa precisione.

- Validation True Negatives (TN): indica il numero di campioni negativi correttamente classificati come tali dal modello, cioè il numero di campioni di uva sana correttamente classificata come tale.

- Validation False Negatives (FN): indica il numero di campioni positivi (anomalie) che il modello ha erroneamente classificato come negativi (invece sani). Un alto valore di FN implica che il modello ha una bassa capacità di richiamare correttamente le anomalie.

- Validation Precision: definita come il rapporto tra i True Positives e la somma di True Positives e False Positives $\left(\frac{TP}{TP+FP}\right)$. Questa metrica misura la capacità del modello di evitare falsi positivi, cioè di non classificare erroneamente uva sana come anomala.

- Validation Recall: definito come il rapporto tra i True Positives e la somma di True Positives e False Negatives $\left(\frac{TP}{TP+FN}\right)$. Il recall misura la capacità del modello di identificare correttamente tutte le anomalie presenti, evitando falsi negativi.

- Validation F1 Score: è la media armonica tra Precision e Recall e fornisce una valutazione più bilanciata delle prestazioni del modello, specialmente in presenza di classi sbilanciate. Un buon F1 score implica che il modello ha sia un alto livello di Precision che di Recall.

- Confusion Matrix: è una tabella che confronta le previsioni di un modello di classificazione con le etichette reali. Mostra quante volte il modello ha previsto

correttamente o erroneamente ciascuna classe, evidenziando i falsi positivi, i falsi negativi, e le previsioni corrette. Aiuta a valutare in modo dettagliato le prestazioni del modello

Queste metriche, combinate tra loro, forniscono una valutazione completa delle prestazioni del modello, sia in termini di accuratezza che di capacità di distinguere correttamente tra le due classi di interesse.

6.4 Validazione e iperparametri

Il modello è stato allenato e validato su epochs e learning rates diversi, basandosi sulle metriche appena descritte per la scelta degli iperparametri da utilizzare per la fase di testing. Di seguito uno snippet di codice:

Listing 6.3. training e validation

```
epochs_range = [50, 100]
learning_rates = [0.001, 0.0005]
best_combination = None
for epochs, lr in product(epochs_range, learning_rates):
    model = SimpleClassifier()
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    train_losses = []
    val_losses = []
    ....others metrics....
    for epoch in range(epochs):
        train_loss = train(model, train_loader, criterion, optimizer)
        train_losses.append(train_loss)
        val_loss, val_accuracy = validate(model, val_loader,
            criterion)
        val_losses.append(val_loss)
        val_accuracies.append(val_accuracy)
        with torch.no_grad():
            model.eval()
            all_preds = []
            all_labels = []
            for inputs, labels in val_loader:
                outputs = model(inputs)
                preds = (outputs > 0.5).float()
                all_preds.extend(preds.cpu().numpy())
                all_labels.extend(labels.cpu().numpy())
            val_f1 = f1_score(all_labels, all_preds)
            ...others metrics...
        val_f1_scores.append(val_f1)
        ... others metrics....
```

```
30      if val_accuracy > best_accuracy or (val_accuracy == best_accuracy
            and val_f1 > best_f1_score):
31          best_accuracy = val_accuracy
32          best_f1_score = val_f1
33          best_combination = (epochs, lr, val_loss, val_accuracy,
                val_f1)
34          results = {
35              'train_losses': train_losses,...
```

La validazione è conclusa producendo i seguenti grafici di metriche

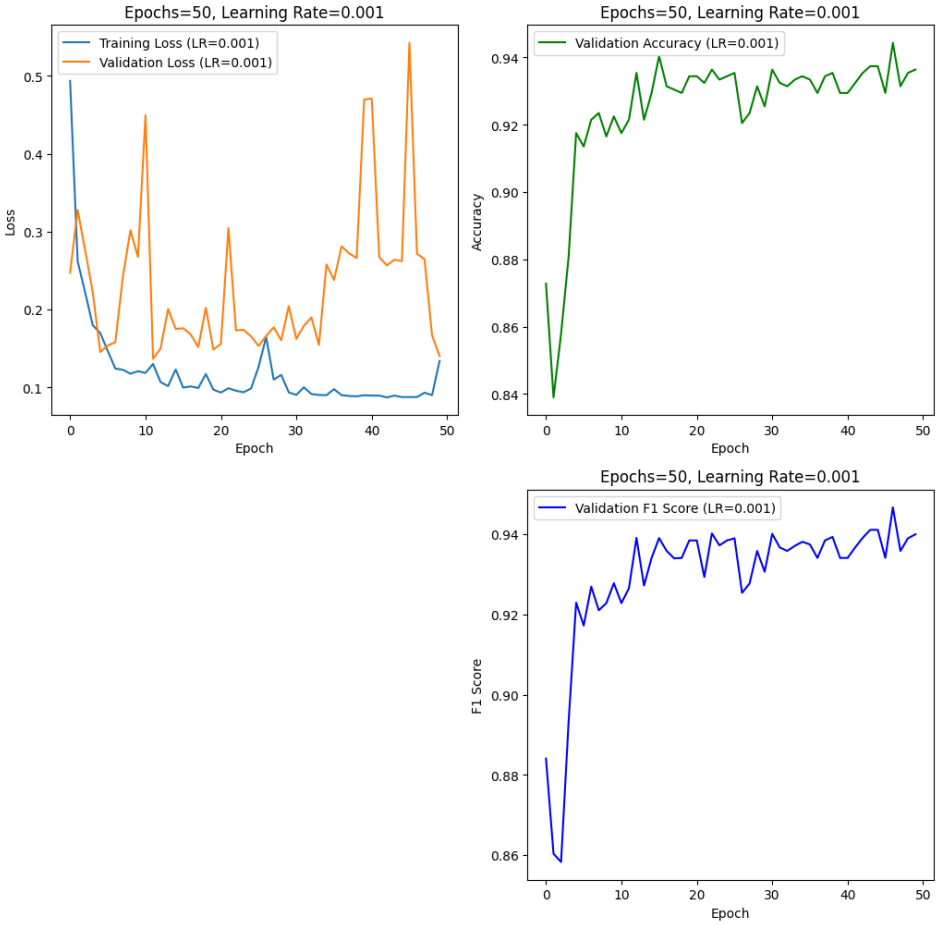

Figura 6.1. metriche prima validazione.

6.5 Risultati dei test - confusion matrix

Infine, il modello è stato provato sui dati di test, producendo i seguenti risultati e la seguente confusion matrix.

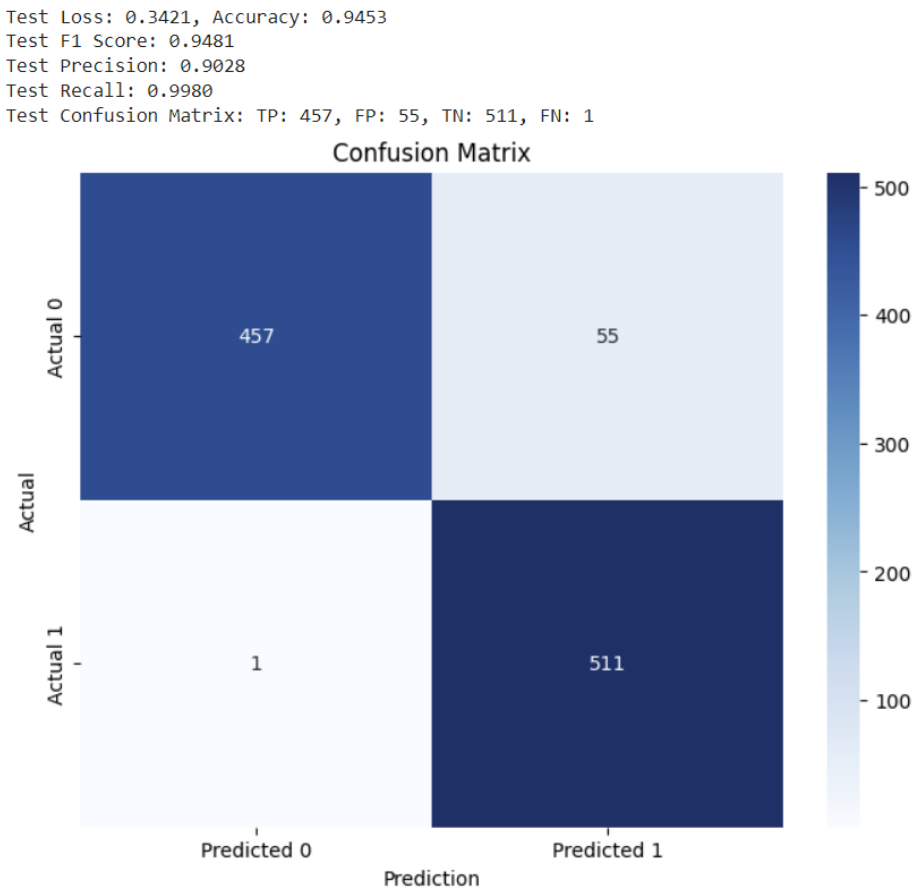

Figura 6.2. metriche e confusion matrix del test.

6.6 Valutazione dei primi risultati

Dalle curve di apprendimento del modello si evince una struttura molto oscillatoria, dove la validation loss non sembra convergere con l'aumentare delle epoche. Se la validation loss continua a oscillare e non migliora con l'aumentare delle epoche potrebbe, ciò indicare la presenza di overfitting, probabilmente derivante dalla struttura della rete neurale utilizzata. Il modello sembra quindi perdere progressivamente la capacità di generalizzare su dati non visti. Per quanto riguarda le metriche, il modello mostra una comprensione più chiara e consistente delle maschere sane, mentre manifesta maggiore incertezza e variabilità quando si tratta di maschere di uva anomala. Ciò si evince dalla matrice di confusione: il numero di falsi negativi è notevolmente maggiore rispetto ai falsi positivi.

Capitolo 7

Modifiche migliorative al classificatore

Per tentare di risolvere i problemi di cui sopra e conseguentemente migliorare le prestazioni del classificatore, ci si appresta ad apportare alcune modifiche al modello Simple Classifier.

7.1 Modifiche alla rete neurale

Nella rete neurale proposta, sono stati introdotti due layer di dropout per migliorare la capacità di generalizzazione del modello e prevenire l'overfitting.
Il dropout è una tecnica di regolarizzazione che disattiva casualmente una percentuale di neuroni durante il training con una probabilità specificata, nel caso presente del 50%.
Nel dettaglio, il dropout è stato applicato come segue.
Dopo il primo layer fully connected (fc1) e la funzione di attivazione ReLU (F.relu), è stato inserito un layer di dropout (self.dropout1). Questo layer disattiva casualmente metà dei neuroni nel passaggio di forward, contribuendo a evitare che il modello dipenda eccessivamente da specifici neuroni e migliorando la generalizzazione. Un secondo layer di dropout (self.dropout2) è stato applicato dopo il secondo layer fully connected (fc2) e la funzione di attivazione ReLU (F.relu). Anche in questo caso, la disattivazione casuale di metà dei neuroni contribuisce a costruire una rete neurale più robusta.
L'implementazione del dropout aiuta a ridurre il rischio di overfitting, costringendo il modello a sviluppare rappresentazioni più generali e a non dipendere eccessivamente da particolari neuroni. Questa tecnica di regolarizzazione è efficace nel migliorare la performance del modello sui dati di validazione e test, assicurando una migliore capacità di generalizzazione.

Di seguito lo snippet di codice.

Listing 7.1. Simple classifier modificato

```
class SimpleClassifier(nn.Module):
    def __init__(self):
        super(SimpleClassifier, self).__init__()
        self.fc1 = nn.Linear(7 * 7 * 512, 512)
        self.dropout1 = nn.Dropout(0.5)            #dropout
        self.fc2 = nn.Linear(512, 128)
        self.dropout2 = nn.Dropout(0.5)            #dropout
        self.fc3 = nn.Linear(128, 1)
    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = self.dropout1(x)                       #dropout
        x = F.relu(self.fc2(x))
        x = self.dropout2(x)                       #dropout
        x = torch.sigmoid(self.fc3(x))
        return x
```

7.2 Test con modello modificato

Il modello è stato allenato e validato nuovamente con le medesime funzioni sugli stessi dati, ottenendo le seguenti curve di apprendimento con i migliori parametri.

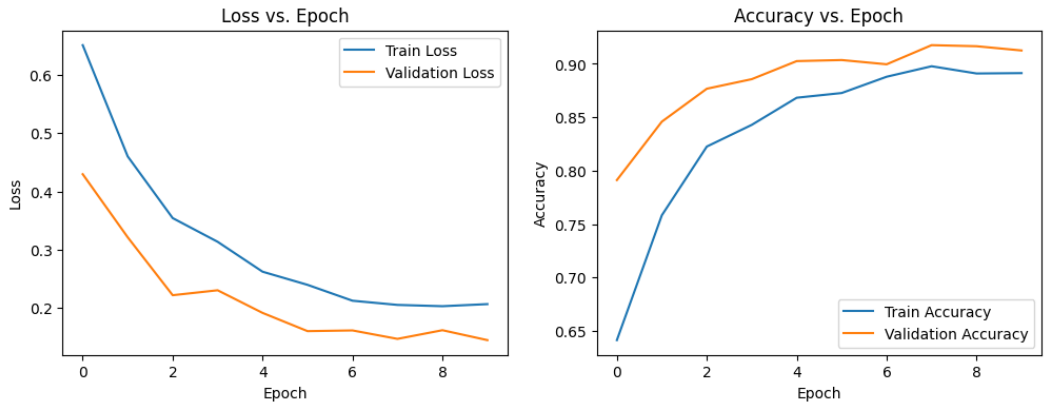

Figura 7.1. curve di apprendimento modificate.

Infine è stata eseguita la funzione di test, producendo i seguenti risultati.

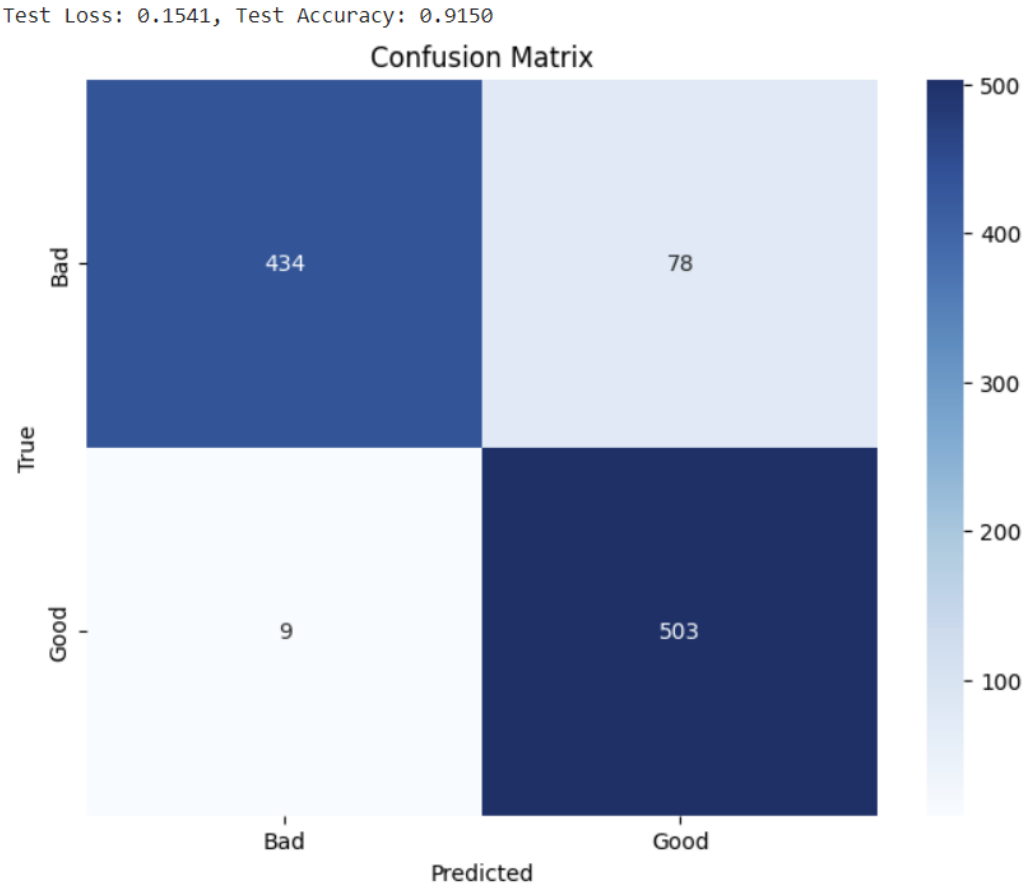

Figura 7.2. confusion matrix del secondo test.

7.3 Considerazioni ulteriori

Si nota come la rete neurale modificata abbia migliorato sensibilmente le prestazioni del modello. Le curve di apprendimento appaiono linearmente più stabili: le losses convergono con l'avanzare delle epoche verso valori molto bassi, mentre l'accuratezza aumenta progressivamente. Per quanto riguarda le metriche e la matrice di confusione, si registra un esiguo aumento dei falsi positivi e dei falsi negativi, con una conseguente diminuzione dell'accuratezza. Tuttavia, la perdita di una percentuale di accuratezza è giustificata dal miglioramento delle curve di apprendimento, che suggerisce un miglior funzionamento complessivo del modello. Anche se l'accuratezza risulta inferiore del 2-3%, il miglioramento nella stabilità delle curve di apprendimento indica che il modello potrebbe essere più robusto e generalizzare meglio sui dati non visti.

Capitolo 8

Conclusioni

In questa tesi sono state sviluppate e sperimentate tecniche avanzate di anomaly detection applicate all'analisi di immagini di uva da tavola, utilizzando reti neurali profonde e metodi innovativi di segmentazione come Grounded Segment Anything (GSAM). Il sistema proposto ha dimostrato la capacità di identificare e segmentare con successo le anomalie presenti nei grappoli d'uva, dimostrando come l'approccio integrato tra segmentazione e classificazione possa fornire soluzioni efficaci a problemi complessi nel contesto agricolo.

L'integrazione tra Grounding Dino, per il rilevamento degli oggetti, e Segment Anything, per la segmentazione delle immagini, ha rappresentato un significativo miglioramento nella fase di pre-processamento, consentendo un'estrazione accurata delle features. Questo ha successivamente permesso di migliorare le prestazioni del modello di classificazione applicato alle immagini di uva sana e anomala.

Le modifiche apportate alla struttura della rete neurale con l'introduzione di layer di dropout, hanno contribuito a ridurre il rischio di overfitting e a migliorare la capacità del modello di generalizzare su dati non visti. Nonostante una piccola diminuzione dell'accuratezza finale, l'analisi delle curve di apprendimento suggerisce che il modello abbia acquisito una capacità più robusta di generalizzazione, divenendo più affidabile in scenari di applicazione reale.

In conclusione, questo lavoro dimostra che tecniche avanzate di machine learning e deep learning possono essere applicate con successo all'anomaly detection nel settore agricolo. Le prospettive future includono l'ulteriore ottimizzazione dei parametri del modello, l'integrazione con altre metodologie di apprendimento automatico e l'estensione del sistema ad altre colture per il monitoraggio e l'identificazione delle anomalie.

Questo approccio rappresenta un passo avanti verso l'automazione nell'agricoltura di precisione, con rilevanti implicazioni per l'efficienza e la sostenibilità delle pratiche agricole.

Bibliografia

[1] Immagine presa da: https://medium.com/@saiwadotai/anomaly-detection-3b6a6bc79a64

[2] Immagine presa da: https://medium.com/@saiwadotai/anomaly-detection-3b6a6bc79a64

[3] Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Qing Jiang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, Lei Zhang, "*Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection*", 9 Mar 2023

[4] Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Qing Jiang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, Lei Zhang, "*Grounding DINO: Marrying DINO with Grounded Pre-Training for Open-Set Object Detection*", 9 Mar 2023

[5] Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg ,Wan-Yen Lo, Piotr Dollar´,Ross Girshick, "*Segment Anything*", 5 Apr 2023

[6] International Digital Economy Academy (IDEA) Community, *"Grounded SAM: Assembling Open-World Models for Diverse Visual Tasks"*, 25 Gen 2024

[7] Immagine presa da: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

[8] Immagine presa da: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

[9] Immagine presa da: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

[10] Immagine presa da: https://www.ibm.com/topics/convolutional-neural-networks

[11] Immagine presa da: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

[12] Immagine presa da: ttps://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[13] Immagine presa da: https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5918

[14] Immagine presa da: https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5918

[15] Karen Simonyan, Andrew Zisserman, "*VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION*", Visual Geometry Group, Department of Engineering Science, University of Oxford, 4 Sett 2014.

[16] https://www.ibm.com/topics/anomalydetection#:~:text=Anomaly%20detection%2C%20or%20outlier%20detection,rest%20of%20a%20data%20set.

[17] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-nn

[18] https://it.mathworks.com/videos/introduction-to-deep-learning

[19] https://medium.com/@saiwadotai/anomaly-detection-3b6a6bc79a64