POLITECNICO DI TORINO

Corso di Laurea Magistrale in
Ingegneria Matematica

Tesi di Laurea Magistrale

Mask R-CNN per la segmentazione di oggetti destinati alla vendita al dettaglio

Relatore
prof. Paolo Garza

Candidato
Pasquale Amara

A.A. 2018/19

Indice

1 Introduzione

I moderni approcci basati sull'utilizzo delle reti neurali hanno permesso la creazione di modelli sempre più accurati per risolvere problemi di regressione e classificazione.
Tra le tematiche affrontate dalle reti neurali convoluzionali, le quali sono un'estensione delle reti neurali artificiali e vengono utilizzate prevalentemente per l'elaborazione di immagini, è possibile citare l'*object detection*, la *semantic segmentation* e l'*instance segmentation*.
Quest'ultimo task racchiude all'interno della sua definizione i task introdotti da classificazione, *object detection* e *semantic segmentation*.
Quando abbiamo un'immagine, un problema di *instance segmentation* consiste nel differenziare ogni singolo oggetto, opportunamente identificato (*object detection*) e classificato al fine di determinarne l'esatta posizione e differenziarlo da qualsiasi altra istanza (anche appartenente alla medesima classe). Quest'ultimo aspetto permette di differenziare un problema di *semantic segmentation* (il quale associerà lo stesso valore a tutte le istanze predette che appartengono alla medesima classe) da un problema di *instance segmentation*.
Alcune applicazioni a problemi di *instance segmentation* riguardano la segmentazione di persone/animali, il riconoscimento e l'individuazione delle aree difettose presenti in oggetti, l'individuazione di cellule tumorali.
All'interno di questo lavoro di tesi è stato sviluppato, in collaborazione con Agile Lab S.r.l., un modello di *instance segmentation* per la segmentazione di oggetti destinati alla vendita al dettaglio. L'utilizzo di questo tipo di modello all'interno di un supermercato può essere molteplice:

- contare il numero esatto di istanze appartenenti ad una determinata classe (ad esempio: contare quanti barattoli sono presenti all'interno dello scaffale);
- analizzare eventuali trend del consumatore analizzando quanti oggetti vengono toccati e/o prelevati;
- analizzare la presenza di oggetti difettosi presenti in uno scaffale

I primi capitoli trattano gli argomenti teorici che sono alla base del funzionamento di Mask R-CNN, ovvero la rete neurale scelta per l'implementazione del modello. In particolare, all'interno del Capitolo 2 verranno introdotte le reti neurali artificiali partendo dall'approccio biologico alla base del loro funzionamento: dal semplice modello del percettrone fino alle reti neurali che costituiscono la base del *deep learning*.
Il Capitolo 3 sarà interamente dedicato alle reti neurali convoluzionali, note anche come CNN, con spiegazione dei layer caratterizzanti questa particolare tipologia di rete.

All'interno del Capitolo 4 viene introdotta Mask R-CNN con uno sguardo alle reti neurali *Region-based* da cui esso trae ispirazione, con uno sguardo alle metriche utilizzate per la valutazione dei modelli ottenuti da Mask R-CNN.
L'implementazione del modello per la segmentazione delle istanze in ambito *retail*, con una spiegazione dettagliata di tutte le fase di training del modello e delle successive valutazioni delle performance, è descritta all'interno del Capitolo 5.
Infine, all'interno del Capitolo 6 verranno commentati i risultati ottenuti, con una valutazione complessiva delle tecniche utilizzate e possibili suggerimenti volti a migliorare le performance ottenute finora.

2 Reti neurali artificiali

2.1 Dalle reti neurali biologiche alle reti neurali artificiali

Ogni rete neurale biologica è composta da circa 10 miliardi neuroni [1], i quali sono composti da un corpo cellulare, chiamato *soma* e da prolungamenti brevi o lunghi, rispettivamente i *dendriti* e gli *assoni*. Al termine di questi ultimi è posibile notare delle ramificazioni, più comunemente note come *sinapsi*.
Il segnale proveniente da un neurone, detto anche *potenziale d'azione* viene propagato attraverso gli assoni: se esso supera una determinata soglia di attivazione, esso si propagherà attivando i neuroni collegati. Questi collegamenti tra i vari neuroni costituiscono una rete, chiamata appunto rete neurale.
Una rete neurale artificiale si basa proprio sui concetti appena descritti: è possibile definirlo come un modello matematico che trae ispirazione dal funzionamento biologico del cervello umano.

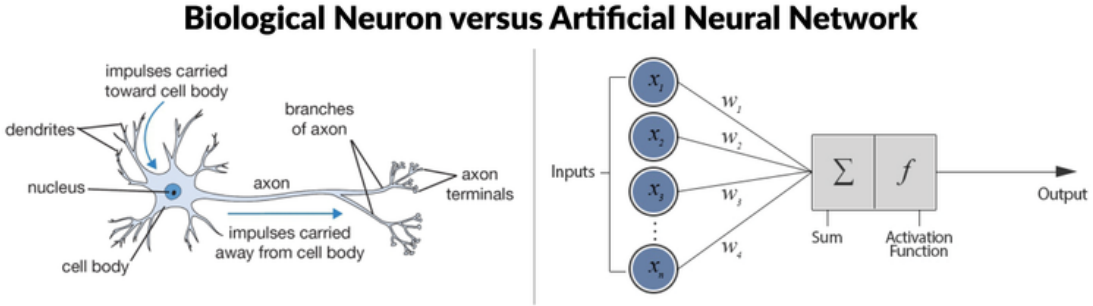


Figura 2.1: Confronto tra rete neurale biologica e rete neurale artificiale [2]

.

In una rete neurale (artificiale) [3], un neurone che riceve informazioni dall'esterno viene detto neurone di input. L'insieme di questi neuroni costituisce il primo strato della rete, detto anche *input layer*. I neuroni appartenenti all'ultimo strato della rete, il cui compito consiste solamente nell'emettere informazioni, ovvero fornire in output una risposta, costituiscono lo strato conclusivo della rete: l'*output layer*. Quando si hanno neuroni che comunicano solamente con neuroni appartenenti alla rete, esse vengono dette unità nascoste. Ciascun neurone della rete si attiverà se la quantità di segnale ricevuta sarà o meno superiore alla propria soglia di attivazione: se ciò avviene,essa trasmetterà l'informazione alle unità ad esso collegato. Questi collegamenti, i

quali simulano il ruolo delle sinapsi, vengono anche detti *pesi*, e verranno indicati con $w_1, ..., w_n$.
E' inoltre doveroso precisare che le reti neurali artificiali vengono utilizzate in primis per risolvere problemi di regressione e classificazione (e, come vedremo all'interno del Capitolo 4, alcuni task più complessi) [4]:

- *Regressione*: in un problema di regressione, dato un valore in input, verrà richiesto all'algoritmo di prevedere un valore numerico in risposta all'input dato. Pertanto, l'obiettivo dell'algoritmo di apprendimento sarà quello di stimare una funzione $f : \mathbb{R}^n \to \mathbb{R}$;
- *Classificazione*: a differenza dei problemi di regressione, un problema di classificazione consiste nell'associare ad ogni valore in input una classe $k \in \{1, .., n\}$. Pertanto, l'obiettivo dell'algoritmo sarà quello di stimare una funzione $f : \mathbb{R}^n \to \{1, ..., n\}$.

2.1.1 Il percettrone

Un modello semplice di rete neurale artificiale è rappresentato dal percettrone (*perceptron*).

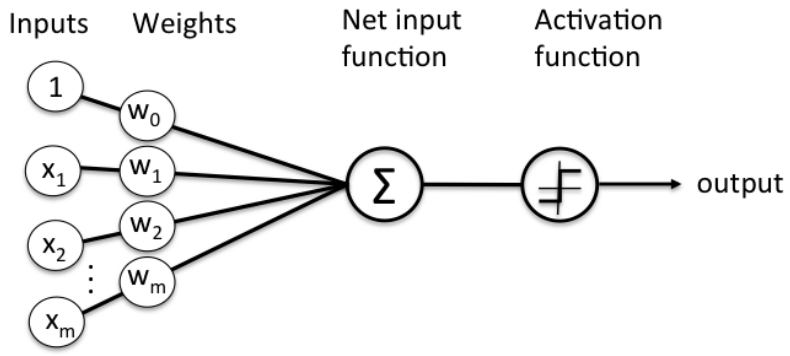


Figura 2.2: Architettura di rete del percettrone di Ronsenblatt

Introdotto da Rosenblatt tra gli anni '50 e gli anni '60 per risolvere un problema di classificazione in due classi, il percettrone riceve in input diversi elementi $x_1, x_2, ..., x_n$ per poi restituire in output un ulteriore segnale binario il cui valore sarà uguale a:

$$output = \begin{cases} 0: & \sum_j w_j x_j \leq threshold \\ 1: & \sum_j w_j x_j > threshold \end{cases} \tag{2.1}$$

Il termine all'interno della sommatoria si può scrivere in maniera compatta come $w \cdot x = \sum_j w_j x_j$. Inoltre, spostando $threshold$ a sinistra, è possibile introdurre un nuovo termine detto *bias*, indicato comunemente con b, il quale sarà definito come $b \equiv -threshold$.
Quindi è possibile riscrivere (2.1) come:

$$output = \begin{cases} 0: & w \cdot x + b \leq 0 \\ 1: & w \cdot x + b > 0 \end{cases} \tag{2.2}$$

Rosenblatt dimostrò che l'algoritmo di apprendimento del percettrone converge se le due classi sono linearmente separabili, ovvero se esiste una iperpiano lineare che separi correttamente le due classi.
In generale, un perceptron si compone di due strati: il primo, detto anche *input layer*, ha dimensione uguale al numero di elementi in input dati dalla rete, detti anche neuroni di input. Ciascun nodo appartenente al primo strato è collegato al nodo di output, il quale costituisce la risposta che la rete neurale elabora per ciascun elemento dato in input. Si può quindi dire che il percettrone proposto da Rosenblatt è una rete neurale avente solamente due strati: uno di input ed uno di output.

Algoritmo del percettrone

L'algoritmo di appprendimento [5] del percettrone è il seguente:

1. **Inizializzazione pesi e bias**: al tempo $t = 0$ vengono inizializzati pesi $(w_i(0))$ e bias (b_i) in maniera casuale $\forall i = 1, ..., n$.

2. **Attivazione**: viene presentato un vettore di elementi di input $x = [x_1, ..., x_n]$ con il corrispondente output $y = [y_1, .., y_n]$

3. **Calcolo dell'output**: l'output corrente verrà calcolato secondo la formula (2.1)

4. **Aggiornamento**: i pesi vengono modificati secondo la *delta rule*, ovvero:

$$w_i(t+1) = w_i(t) + \eta \cdot (y - a(t)) \cdot x_i(t) \tag{2.3}$$

   dove $a(t)$ è l'output calcolato al punto 3 e η è un valore compreso tra 0 e 1 chiamato anche *learning rate*.

5. **Continuazione**: la procedura si ripete tornando al punto 2.

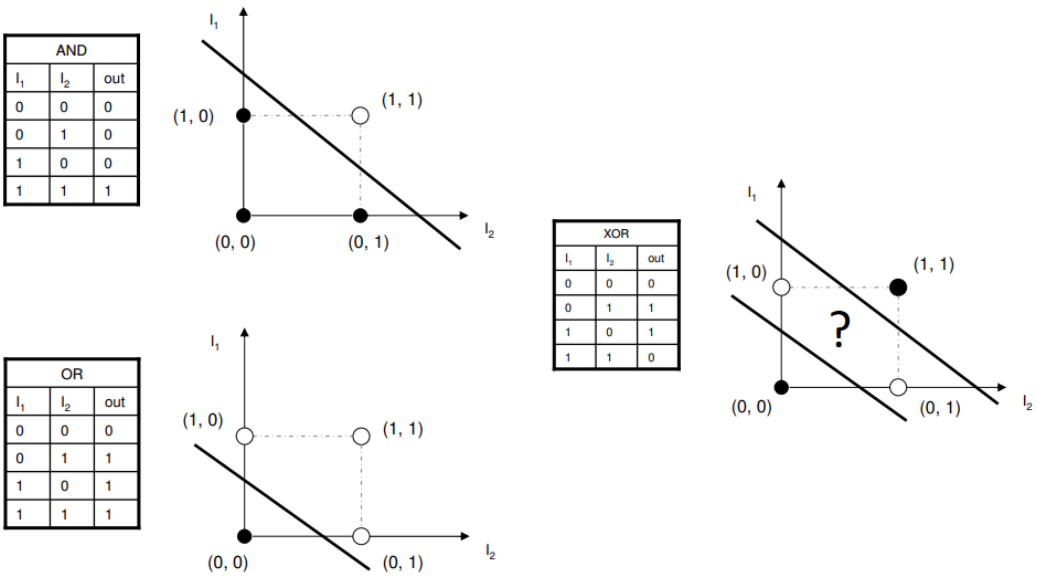


Figura 2.3: Problemi di classificazione binaria con relativa rappresentazione grafica [6].

L'algoritmo del percettrone verrà ripetuto fino a quando tutti gli elementi verranno classificati correttamente (o per un numero N di iterazioni dichiarato inizialmente). Di seguito verranno mostrati degli esempi di classificazione svolti dall'algoritmo del percettrone per i problemi logici AND, OR e XOR. Graficamente, la retta di decisione determinata dal percettrone riesce a classificare in maniera corretta ciascun elemento in input nel caso dei problemi AND e OR. A differenza di ultimi, è possibile notare come una retta non sia sufficiente per separare le due classi nel problema XOR [7]: ciò è dovuto al fatto che tale rete riesce a classificare correttamente due classi linearmente separabili, ma non riesce a risolvere un problema di classificazione tra classi non linearmente separabili, quale ad esempio il problema XOR.

2.1.2 Architettura di un MLP

Il percettrone fu accantonato proprio per i limiti di classificazione precedentemente descritti.
Per risolvere il problema dell'XOR è necessaria una rete neurale avente uno o più strati compresi tra l'*input layer* e l'*output layer*. Un modello di rete neurale di questo tipo è anche detto percettrone multistrato (*multi layer perceptron*), o anche rete neurale *feedforward*, poichè l'informazione fluisce dai nodi appartenenti all'input layer fino ai nodi dell'output layer senza scambio di informazioni tra nodi dello stesso strato.

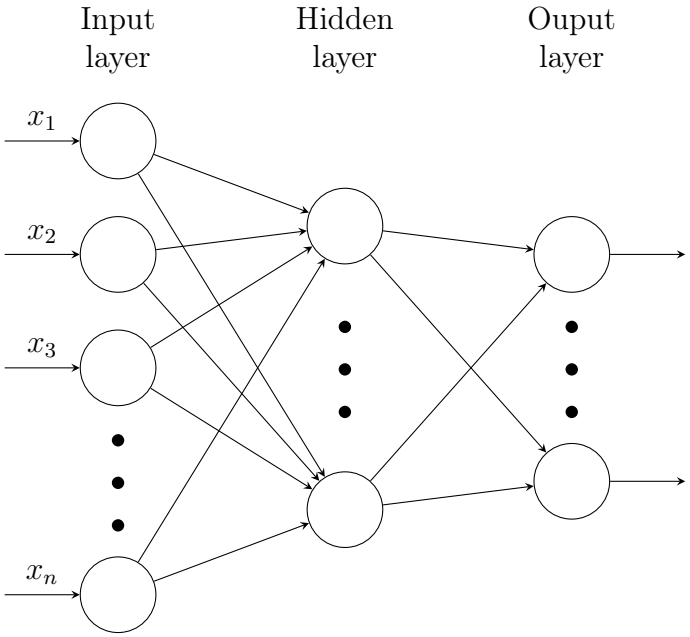


Figura 2.4: Esempio di rete neurale avente un solo hidden layer.

La figura 2.4 mostra una rete neurale avente quattro nodi di input: l'insieme di questi nodi costituisce il primo strato della rete, ovvero *input layer*.
Ogni neurone in input è connesso a tutti i neuroni presenti all'interno dello strato intermedio, chiamato anche *hidden layer* poichè contenente solamente neuroni non appartenenti al primo e all'ultimo strato. Questo tipo di rete è detta anche *fully-connected network*, poichè ciascun neurone è collegato a tutti i neuroni dello strato precedente, ma due neuroni appartenenti allo stesso layer non possono essere connessi tra loro.
Nel corso dell'ultimo decennio è diventato sempre più frequente l'utilizzo di reti neurali aventi molteplici *hidden layer*. Una rete neurale avente più di un *hidden layer* è detta anche rete neurale profonda e l'insieme di queste reti costituiscono la base del moderno *deep learning*.
Rispetto alla precedente notazione utilizzata per descrivere il percettrone, nel caso di una *deep neural network* denoteremo pesi, bias e funzione di attivazione con la seguente terminologia:

- w^l_{jk} è il peso per la connessione tra il k-esimo neurone dell' $(l-1)$ esimo layer e il j-esimo neurone dell' l-esimo layer
- b^l_j è il j-esimo bias dell' l-esimo layer
- a^l_j è la j-esima funzione di attivazione dell' l-esimo layer, con

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) \tag{2.4}$$

  dove la somma è su tutti i k neuroni dell'l-esimo layer.
  Spesso la quantità $\sum_k w^l_{jk} a^{l-1}_k + b^l_j$ è indicata semplicemente come z^l, la quale rappresenterà la somma pesata su tutti i neuroni dell' l-esimo layer.
  Ricapitolando, ponendo $z \equiv \sum_k w^l_{jk} a^{l-1}_k + b^l_j$, è possibile riscrivere l'equazione (2.4) come:

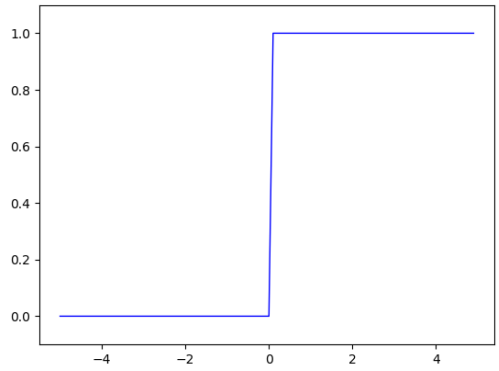
$$a^l_j = \sigma(z^l) \tag{2.5}$$

2.1.3 Funzioni di attivazione

La scelta di un'opportuna funzione di attivazione assume un ruolo molto importante: l'output restituito da una rete neurale è fortemente condizionato dalla funzione di attivazione utilizzata, la quale possiede un ruolo molto importante anche nel processo e nella velocità di convergenza della rete neurale e nella sua accuratezza.
Inoltre, nelle moderni reti neurali profonde, la scelta della funzione d'attivazione da utilizzare dipende dal tipo di layer a cui essa è associato. Nel corso degli anni sono state utilizzate diverse funzioni di attivazione: binarie, lineari e non lineari. Diamo un'occhiata alle più comuni:

Step function

La step function σ_{step} è definita come:



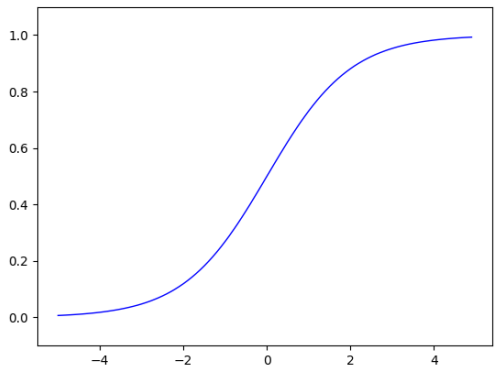
$$\sigma(z)_{step} = \begin{cases} 1 & z \geq 0 \\ 0 & x < 0 \end{cases} \tag{2.6}$$

Figura 2.5: Grafico della step function

Non viene molto utilizzata in quanto si perdono le informazioni quantitative relative all'input (ad esempio, $\sigma_{step}(0.5) = \sigma_{step}(1000) = 1$).

Funzione Sigmoide

La funzione sigmoide $\sigma_{sigmoid}$ è definita come:



$$\sigma_{sigmoid}(z) = \frac{1}{1 - e^{-z}} \tag{2.7}$$

Figura 2.6: Grafico della funzione sigmoide

Poichè la funzione sigmoide restituisce in output un valore appartenente all'intervallo [0,1], essa viene utilizzata spesso come funzione di attivazione dell'*output layer* per modelli di classificazione ed il valore restituito in output assume un valore probabilistico se l'output è binario. Qualora si tratta di un problema di classificazione a più classi, esso restituisce la probabilità di

Funzione tangente iperbolica

Un'altra funzione di attivazione utilizzata è la funzione tangente iperbolica (tanh) $\sigma_{\tanh}$, la quale restituisce valori all'interno dell'intervallo $[-1,1]$.

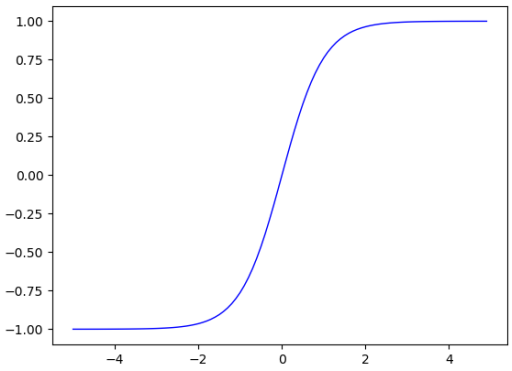


Figura 2.7: Grafico della funzione tanh

$$\sigma_{\tanh}(z) = \frac{e^{2z} - 1}{e^{2z} + 1} \tag{2.8}$$

Funzione reLU (Rectified Linear Unit)

La funzione di attivazione reLU è così definita:

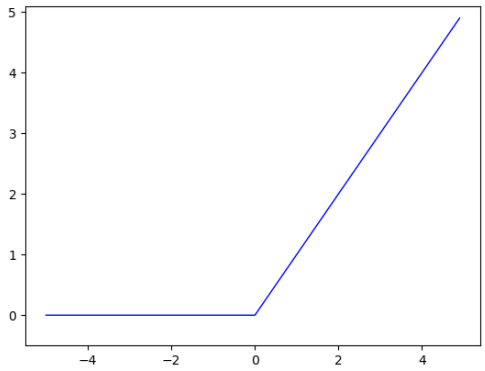


Figura 2.8: Grafico della funzione reLU

$$\sigma_{reLU}(z) = \max\{0, z\} \tag{2.9}$$

Essa sarà quindi uguale a z per $z > 0$, mentre sarà nulla per tutti gli altri valori.
E' la funzione di attivazione maggiormente utilizzata nelle reti neurali artificiali e ha sostituito nel corso degli anni le funzioni tangente iperbolica e sigmoide. Il motivo principale è dovuto al fatto che $\phi_{\tanh}$ e $\phi_{sigmoid}$ hanno una derivata prima molto piccola, la quale tende velocemente a 0.

Poichè il training di una rete neurale si basa sulla discesa del gradiente, la moltiplicazione per un valore prossimo a 0 porta i layer più profondi della rete ad un apprendimento più lento (come vedremo più avanti introducendo la *backpropagation*, se il gradiente dei neuroni di output verrà moltiplicato per un valore pari o prossimo a 0, il segnale che fluirà attraverso i nodi ed i pesi andando all'indietro sarà nullo o prossimo a 0).
E' inoltre molto facile da calcolare (da un punto di vista computazionale, è necessario solamente un confronto per determinarne l'output) e ciò si traduce, nel caso di reti neurali molto profonde, in un notevole risparmio in termini di costo computazionale [8].

Funzione Softmax

Per problemi di classificazione, la funzione di attivazione sigmoide si presta bene nel caso in cui si hanno solamente due classi.
Qualora il numero di classi sia maggiore, si utilizza la funzione di attivazione softmax, così definita:

$$\sigma(z) = \frac{e^{z_j}}{\sum_k e^{z_k}} \tag{2.10}$$

Riscrivendo l'equazione (2.10) utilizzando la notazione introdotta nel paragrafo 2.1.2, si ha:

$$a_j^L = \frac{e^{z_j}}{\sum_k e^{z_k}} \tag{2.11}$$

Le funzioni di attivazione a_j^L relative all'*output layer* di una rete neurale assumono un'interpretazione probabilistica: ciascun a_j^L assume un valore appartenente all'intervallo [0,1] ed inoltre:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j}}{\sum_k e^{z_k}} = 1 \tag{2.12}$$

E' possibile dunque vedere a_j^L come la probabilità stimata dalla rete che l'elemento considerato appartenga alla j-esima classe.

2.2 Training di una rete neurale

Quando viene creata una rete neurale, essa non possiede alcuna forma di conoscenza. Possiamo definire il training di una rete neurale il processo volto a trovare la configurazione dei pesi che massimizzi l'accuratezza del modello [9]. Sono utilizzati differenti approcci per il training di una rete neurale [9] (e, più in generale, un modello di machine learning):

- *Supervised Learning*: tutti i dati presenti all'interno del dataset di addestramento (*training set*) sono etichettati. Pertanto [10], ad ogni osservazione $x_i,\ i = 1, ..., n$ presente all'interno del training set verrà associata una risposta y_i;
- *Unsupervised Learning*: a differenza dell'approccio di tipo supervised, a ciascuna osservazione x_i presente all'interno del training set non risulta associata alcuna risposta y_i;

- *Semi-Supervised Learning*: è un approccio ibrido che coinvolge entrambi gli approcci descritti in precedenza. Esso consiste nell'utilizzare all'interno del training set solamente una piccola porzione di dati etichettati, mentre la restante porzione è composta da dati non etichettati;

- *Reinforcement Learning*: è un differente tipo di approccio in cui, sulla base delle azioni prese dal modello in fase di training, verrà attribuito un premio (se la scelta risulta essere corretta) o una penalità (se la scelta risulta essere errata). Il processo di training consiste, in questo caso, nel determinare il valore dei pesi che massimizzi il premio ottenuto minimizzando il valore associato alla penalità.

Il processo di training consiste nel modificare i pesi della rete affinchè essa riesca ad apprendere gli esempi presenti all'interno del training set mantenendo un'opportuna accuratezza anche in presenza di dati in input non presenti all'interno del training set.

2.2.1 Gradient Descent

Per valutare quanto bene abbia 'imparato' il modello si utilizza la *loss function* J, talvolta indicata anche come *cost function*. In modelli di reti neurali e deep learning si utilizzano comunemente due loss function. Essa rappresenta una misura dell'errore del modello in termini di capacità di stimare la relazione tra un input x e il corrispondente output y [11].
La scelta della loss function da utilizzare dipende dal tipo di problema per cui è stato il modello di rete neurale: per problemi di regressione, la loss function comunemente usata è il MSE (*Mean Squared Error*).

$$J(w,b) = \frac{1}{2n}\sum_{i=1}^{n} \|y(x) - a\|^2 \tag{2.13}$$

dove w e b sono i vettori contenenti, rispettivamente, tutti i pesi e i bias della rete, x è il vettore degli elementi in input avente output y, n è il numero totale di elementi del dataset e a è l'output di x stimato dalla rete.
Al fine di condurre il modello ad apprendere correttamente i dati presenti all'interno del training set è necessario minimizzare la funzione J: per farlo si utilizza l' algoritmo *gradient descent*.
L'idea alla base del gradient descent consiste nel minimizzare la loss function $J(w,b)$ aggiornando il valore di w e b sulla base della differenza tra J ed il suo gradiente negativo calcolato rispetto al parametro considerato [12].
Per comprendere meglio l'idea alla base del *gradient descent*, supponiamo inizialmente che J sia una funzione di due variabili v_1 e v_2, ovvero $J(v_1, v_2)$. La variazione di ciascuna due componenti provocherà una variazione di J esprimibile come:

$$\Delta J \approx \frac{\partial J}{\partial v_1}\Delta v_1 + \frac{\partial J}{\partial v_2}\Delta v_2 \tag{2.14}$$

Per minimizzare J [1], è necessario trovare Δv_1 e Δv_2 tale che ΔJ sia negativo. Definiamo il gradiente di C come il vettore delle sue derivate parziali:

$$\nabla J \equiv \left(\frac{\partial J}{\partial v_1}, \frac{\partial J}{\partial v_2} \right)^T \tag{2.15}$$

e, ponendo $\Delta v = (\Delta v_1, \Delta v_2)^T$, è possibile riscrivere l'equazione (2.14) come:

$$\Delta J \approx \nabla J \cdot \Delta J \tag{2.16}$$

L'importanza dell'equazione (2.16) sta nel fatto che, affinchè J possa decrescere, è necessario scegliere Δv in modo tale che ΔJ sia negativo. In particolare, poniamo:

$$\Delta v = -\eta \cdot \Delta v \tag{2.17}$$

dove η è un numero positivo comunemente noto come *learning rate*. L'equazione (2.14) diventa dunque:

$$\Delta J \approx -\nabla J \cdot \eta \cdot \nabla J = -\eta \|\nabla J\|^2 \tag{2.18}$$

e, poichè $\|\nabla J\|^2 \geq 0$, si ha la certezza che $\Delta J \leq 0$. Questa procedura permetterà di aggiornare le componenti di v che saranno uguali a
:

$$v \to v' = v - \eta \nabla J \tag{2.19}$$

Nel nostro caso, il vettore $\mathbf{v}$ è sostituito da w e b, ovvero i vettori di pesi e bias. Parleremo di *epoca* di training quando ciascun elemento del dataset verrà processato sia in avanti (*forward*) sia all'indietro (*backward*) dalla rete una sola volta.
Per ogni epoca una volta calcolata la loss function J, è necessario dunque derivare J rispetto a w e rispetto a b.
Al termine di ogni epoca verranno infine aggiornati i valori di $\mathbf{w}$ e di $\mathbf{b}$ nel seguente modo:

$$w_i \to w_i' = w_i - \eta \cdot \frac{\partial J}{\partial w_i} \quad \forall i = 1, ..., n \tag{2.20}$$

$$b_i \to b_i' = b_i - \eta \cdot \frac{\partial J}{\partial b_i} \qquad \forall i = 1, ..., n \tag{2.21}$$

Il *learning rate* η regolerà l'aggiornamento di w e b: più grande sarà η, più velocemente l'algoritmo tenderà a convergere verso il punto che minimizza J.
Tuttavia, un learning rate elevato può condurre a degli eccessivi 'salti' all'interno della funzione, il che potrebbe causare una mancanza di convergenza.
Viceversa, la scelta di un learning rate eccessivamente piccolo potrebbe rallentare parecchio il processo di convergenza, rendendo necessario un numero di epoche più elevato al fine di raggiungere risultati accettabili.

E' possibile applicare l'algoritmo del gradient descent calcolando il gradiente sull'intero dataset oppure di selezionare uno o più elementi ad ogni iterazione: viene indicato

[1] Un problema di ottimizzazione su una generica funzione f consiste nel trovare una soluzione che massimizzi (problema di massimizzazione) o minimizzi (problema di minimizzazione) il valore di f (soggetta o meno a determinati vincoli).

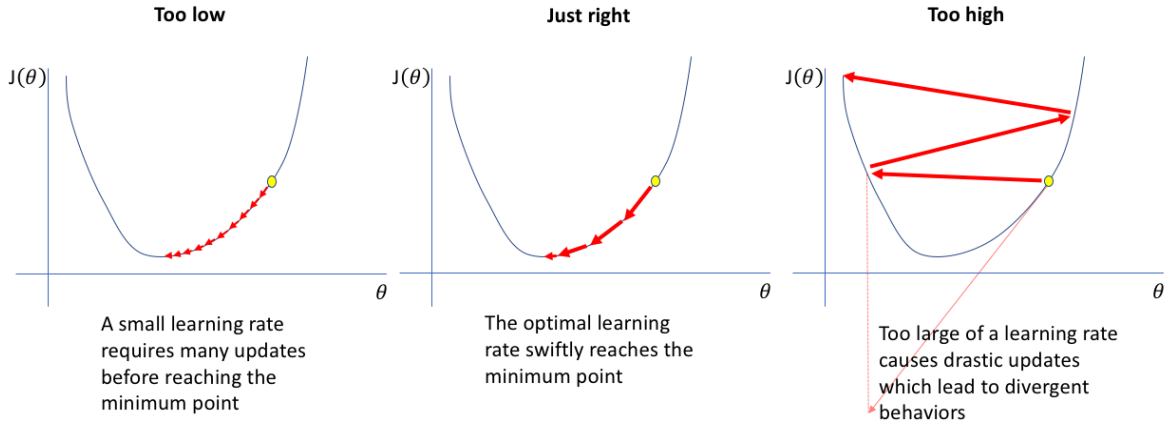


Figura 2.9: I grafici mostrano il differente processo di convergenza verso il punto di minimo globale di J. [13]

col termine *batch* il numero totale di campioni del dataset utilizzati per calcolare il gradiente in una singola operazione. La scelta di un opportuno gruppo di elementi del dataset influisce sull'aggiornamento di w e b, pertanto influisce sul processo di convergenza della loss function.
Poichè l'algoritmo del gradient descent risulta essere una procedura iterativa, risulta abbastanza evidente che effettuare il training di un modello una singola epoca comporterà un solo aggiornamento di pesi e bias. Viceversa, un numero eccessivo di epoche di training può condurre il modello ad adattarsi eccessivamente ai dati di training e, di conseguenza, andare incontro al problema dell'overfitting [14].

Full Batch Gradient Descent

Il *full batch gradient descent* calcola la loss function considerando, ad ogni iterazione, tutti gli elementi presenti all'interno del dataset: J sarà dunque la loss function media calcolata sull'intero dataset, ovvero $J = \frac{1}{n}\sum_i J_i$. La procedura di aggiornamento di $\mathbf{w}$ e $\mathbf{b}$ è la medesima descritta dalle equazioni (2.44) e (2.43). Nonostante la semplicità a livello intuitivo della procedura, essa comporta diversi svantaggi: poichè vengono utilizzati tutti gli elementi presenti all'interno del dataset ad ogni singola iterazione, tale operazione risulta essere parecchio onerosa in presenza di dataset di grandi dimensioni. Un altro problema consiste nella sua scarsa dinamicità: per migliorare il modello con nuovi dati è necessario ripetere il processo di training sull'intero dataset.
Inoltre, tale procedura risulta essere molto soggetta al problema dei minimi locali.

2.2.2 Stochastic Gradient Descent

Uno dei problemi del *full batch gradient descent* riguardava la necessità di calcolare ∇J su tutti gli elementi del dataset, a prescindere dalla dimensione di esso.
L'approccio dello *stochastic gradient descent* è differente: ∇J non è più la media sui ∇J calcolati sull'intero dataset, bensì viene stimato da un singolo elemento del dataset scelto in maniera casuale.
A differenza del precedente approccio, la scelta di un singolo elemento del dataset per

il calcolo di J porta ad un notevole miglioramento in termini di tempo d'esecuzione dell'algoritmo. Da un punto di vista grafico, è possibile che J tenda parecchio ad oscillare: tali oscillazioni potrebbero condurre la loss function ad uscire più facilmente da punti di minimo locale.

Mini Batch Gradient Descent

Lo *stochastic gradient descent* riesce tuttavia a risolvere solo parzialmente il problema dei minimi locali.
Una via di mezzo tra le tecniche precedentemente descritte è il *mini batch gradient descent*: all'interno del dataset, ad ogni epoca viene estratto un sottoinsieme di elementi del dataset e la loss function calcolata è la media delle loss functions calcolate all'interno del sottoinsieme. Pertanto, scelto un numero $m < n$ come batch size, ∇J sarà uguale a:

$$\nabla J = \frac{1}{n}\sum_{i} \nabla J_i \approx \frac{1}{m}\sum_{j=1}^{m} \nabla J_j \tag{2.22}$$

e le equazioni (2.44) e (2.43) diventeranno:

$$w_i \to w_i' = w_i - \frac{\eta}{m} \cdot \frac{\partial J_{x_i}}{\partial w_i} \quad \forall i = 1, ..., n \tag{2.23}$$

$$b_i \to b_i' = b_i - \frac{\eta}{m} \cdot \frac{\partial J_{x_i}}{\partial b_i} \quad \forall i = 1, ..., n \tag{2.24}$$

All'interno della figura 2.10 è rappresentato il conronto tra le traiettorie prodotte dall tre tecniche citate: è possibile notare come lo *stochastic gradient descent* e il *mini batch gradient descent* producano traiettorie che tendano ad andare "a zig-zag": ciò è dovuto al fatto che vengono scelti, rispettivamente un elemento ed un sottocampione dal dataset di partenza. Pertanto, a differenza del *full batch gradient descent*, non stiamo calcolando il gradiente di J verà e proprio, bensì una sua approssimazione.

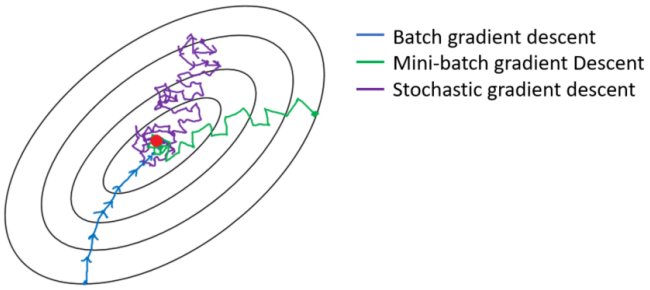


Figura 2.10: Confronto tra le traiettorie prodotte dagli algoritmi basati sul gradient descent [15]

2.2.3 Backpropagation

Il processo di calcolo delle predizioni che parte dai coefficienti per poi terminare con l'errore è noto come *forward propagation*.
Viceversa, il processo che permette di ottimizzare i coefficienti w e b partendo dall'errore calcolato precedentemente viene detto *backward propagation* o, più in generale, *backpropagation*. Questo processo, introdotto già nel 1970, acquisì notevole importanza nel 1986, anno in cui D. Rumelhart, G. Hinton e R. Williams [16] evidenziarono quanto una rete neurale apprenda più velocemente grazie alla backpropagation rispetto alle tecniche utilizzate in precedenza. Ancora oggi, esso assume un'importanza notevole al fine dell'apprendimento delle più moderne reti neurali profonde.
Per comprenderne il funzionamento [17], aggiungiamo a $\frac{\partial J}{\partial w}$ e $\frac{\partial J}{\partial b}$ un termine di errore δ_j^l, ovvero l'errore presente nel neurone j-esimo del layer l-esimo, z_j^l. L'aggiunta di tale termine di errore al neurone z_j^l porterà alla modifica della funzione di attivazione associata ad esso, la quale sarà adesso uguale a $\phi(z_j^l + \Delta_j^l)$ propagandosi all'interno della rete e causando così un'altrettanto modifica alla loss function totale, la quale subirà una variazione di $\frac{\partial J}{\partial z_j^l}\Delta z_j^l$.
E' possibile dunque monitorare la variazione della loss function facendo variare il termine di errore δ_j^l, definito come

$$\delta_j^l = \frac{\partial J}{\partial z_j^l} \tag{2.25}$$

La backpropagation si basa principalmente su quattro equazioni, le quali costituiscono la base per il calcolo del gradiente di J e del termine di errore δ:

- Equazione per l'errore all'interno dell'output layer σ^L:

$$\delta_j^L = \frac{\partial J}{\partial a_j^L}\sigma'(z_j^L) \tag{2.26}$$

- Equazione per l'errore δ^l espresso come errore del layer successivo:

$$\delta^l = ((w^{l+1})) \odot \sigma'(z^L) \tag{2.27}$$

- Equazione per la variazione della loss function rispetto a qualsiasi bias:

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l \tag{2.28}$$

- Equazione per la variazione della loss function rispetto a qualsiasi peso:

$$\frac{\partial J}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{2.29}$$

L'equazione (2.26) mostra come il calcolo del termine di errore venga calcolato sull'output layer, per poi essere propagato nella rete all'indietro (2.27) fino all'input layer.
Le equazioni (2.28) e (2.29) mostrano le relazioni tra δ e le derivate parziali calcolate

rispetto a w e b.
Nella pratica è bene combinare l'algoritmo di *backpropagation* con un'algoritmo basato sul *gradient descent*. Supponendo di utilizzare un *mini batch gradient descent*, con batch size di $m < n$ elementi (n dimensione del dataset), è possibile vedere come l'azione combinata dei due algoritmi agisce sul processo di training:

1. Selezione casuale di m valori di input all'interno del dataset
2. $\forall x_i \in x_1, x_2, .., x_m$:
   - **Feedforward**: $z^{x,l} = w^l a^{x,l-1} + b^l, \quad a^{x,l} = \phi(z^{x,l}) \quad \forall l = 2,3, ..., L$
   - **Output error** $\delta^{x,L}$: calcolo del vettore $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$
   - **Backpropagate the error**: $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$ $\forall l = L-1, L-2, ...,2$
3. *Gradient descent*: $\forall l = L-1, L-2, ...,2$:
   - aggiornamento di w: $w_i \to w_i' = w_i - \frac{\eta}{m} \cdot \sum_x \delta^{x,l} (a^{x,l-1})^T$
   - aggiornamento di $b_i \to b_i' = b_i - \frac{\eta}{m} \cdot \sum_x \delta^{x,l}$

2.2.4 Cross-entropy

La loss function più usata per problemi di regressione è la già descritta *Mean Squared Error* (MSE).
Supponiamo, per semplicità, di avere una rete neurale composta da un solo input layer ed un solo output layer, e di dover classificare un elemento avente input x=1 e output $y = 0$. Utilizzando come loss function la *MSE*, si avrà:

$$J = \frac{1}{2}(y - a)^2 \tag{2.30}$$

le cui derivate parziali saranno uguali a:

$$\frac{\partial J}{\partial w} = (a - y)\sigma'(z)x = a\sigma'(z) \tag{2.31}$$

$$\frac{\partial J}{\partial b} = (a - y)\sigma'(z)x = a\sigma'(z) \tag{2.32}$$

Utilizzando una funzione di attivazione softmax, è possibile vedere dal grafico 2.1.3 come la sua derivata prima tenda ad assumere valori molto piccoli in prossimità dei valori estremi 0 e 1: ciò causa un rallentamento nel processo di apprendimento della rete.
Per cercare di risolvere il problema relativo alla lentezza in fase di apprendimento, per problemi di classificazione la loss function maggiormente utilizzata risulta essere la binary cross-entropy, così definita:

$$J = -\frac{1}{n} \sum_x [y \ln a + (1 - y) \ln(1 - a)] \tag{2.33}$$

e le sue derivate parziali, dopo notevoli semplificazioni, saranno uguali a:

$$\frac{\partial J}{\partial w} = \frac{1}{n} \sum_x x_j(\sigma(z) - y) \tag{2.34}$$

$$\frac{\partial J}{\partial b} = \frac{1}{n} \sum_x (\sigma(z) - y) \tag{2.35}$$

Generalizzando l'equazione (2.33) per le *deep neural networks* nel caso di un problema di classficazione binario, si ha:

$$J = -\frac{1}{n} \sum_x [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)] \tag{2.36}$$

mentre, per problemi di classificazione su più classi si avrà:

$$J = \sum_x y_j \ln a_j^L \tag{2.37}$$

2.3 Overfitting

In generale, un qualunque modello di machine learning apprende la propria conoscenza dagli esempi forniti all'interno del training set.
Come già è stato visto, la loss function è una misura di quanto il modello abbia imparato bene i dati appresi. Questa misura dell'errore è scomponibile in due parti: errore irriducibile ed errore riducibile. Il primo, definito come quell'errore che non può essere ridotto indipendentemente dall'algoritmo applicato, non può essere eliminato e sarà sempre presente all'interno della loss function. Viceversa, è possibile minimizzare l'errore riducibile, il quale è composto da due componenti [18]: l'errore dovuto al bias e l'errore dovuto alla varianza.

- *errore dovuto al bias*: è dato dalla differenza tra la predizione del modello ed il valore che stiamo cercando di predire. Quando il suo valore è elevato, si dice che il modello sta andando incontro al problema dell'*underfitting*;
- *errore dovuto alla varianza*: misura la variabilità della predizione su certi dati. Un valore elevato indica la difficoltà del modello nell'effettuare predizioni corrette su dati per cui non è stato allenato.

All'interno della figura 2.11 sono mostrati tre diversi modelli di regressione: il primo non riesce ad adattarsi sufficientemente ai dati (elevato bias), pertanto è soggetto al problema dell'*underfitting*. Viceversa, il terzo esempio mostra un modello con elevata varianza: esso cercherà di adattarsi a tutti i dati presenti. Ne verrà fuori un modello eccessivamente complesso che farà fatica ad adattarsi a dati mai visti in precedenza.

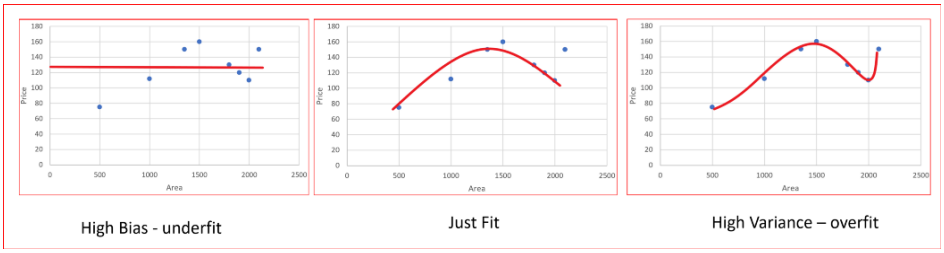


Figura 2.11: Esempio di modelli di regressione. Il primo ed il terzo sono soggetti, rispettivamente, al problema dell'underfitting e dell'overfitting [18].

Quando un modello è soggetto al problema dell'*overfitting* si fa riferimento ad un modello che ha appreso eccessivamente gli esempi proposti all'interno del training set, non riuscendo a garantire performance soddisfacenti nella valutazione di ulteriori dati. Apprendere eccessivamente gli esempi presenti nel training set comprta l'aver imparato i dettagli ed il rumore presenti all'interno di ciascun elemento del suddetto dataset. Un primo approccio per evitare il problema dell'overfitting consiste nel valutare le performance finali del modello su un *test set*. Tuttavia, risulta più efficiente l'utilizzo di un ulteriore dataset, il *validation set* per valutare le performance del modello alla fine di ogni epoca di training: ciò permette di monitorare l'andamento dell'apprendimento e ci dà un'idea di quanto il nostro modello stia imparando a generalizzare i concetti appresi.

Non è facile valutare il perchè un modello tenda ad andare in overfitting: ciò può dipendere dal valore degli iperparametri in fase di apprendimento, dal tipo di rete utilizzata, dal numero eccessivo di epoche di training, da esempi troppo simili e/o generici presenti nel training set.
Risulta quindi evidente che è necessario l'utilizzo di tecniche che portino a migliorare la capacità di generalizzazione (riuscire ad apprendere su un sottoinsieme dei dati e di riuscire a classificare correttamenti esempi esterni ai casi già presenti in fase di training) del modello.

2.4 Regolarizzazione

Una prima tecnica utilizzata per ridurre il problema dell'overfitting di un modello è la regolarizzazione. E' possibile definirla come "qualsiasi modifica apportata ad un algoritmo di apprendimento che permetta la riduzione del suo errore di generalizzazione ma non del suo errore di training" [4].
Denotiamo con $\tilde{J}$ la funzione obiettivo regolarizzata:

$$\tilde{J} = J + \lambda\Omega \tag{2.38}$$

dove con J indichiamo la funzione obiettivo da minimizzare nel processo di training e con $\lambda \in [0, \infty]$ l'iperparametro che regola l'impatto della norma Ω su J (un valore elevato di λ indica un'elevata regolarizzazione, mentre $\lambda = 0$ ne indica un contributo nullo).
I risultati possono variare a seconda della scelta di Ω: vediamo quali sono le tecniche di regolarizzazione più utilizzate.

2.4.1 Regolarizzazione L_2

La metricha di regolarizzazione maggiormente utilizzata è la regolarizzazione in norma L_2, spesso indicata come *weight decay*. Essa consiste nell'aggiunta del termine

$$\Omega = \frac{1}{2n} \sum_x w^2 \tag{2.39}$$

alla loss function J. Si otterrà dunque una nuova loss function $\tilde{J}$ pari a

$$\tilde{J} = J + \frac{\lambda}{2n} \sum_x w^2 \tag{2.40}$$

L'utilizzo della regolarizzazione L_2 penalizza i pesi più grandi della rete limitandone l'impatto sul modello. Al contrario, al fine di minimizzare $\tilde{J}$, vengono accettati valori di w molto piccoli. Maggiore sarà il valore di λ, maggiore la rete accetterà bassi valori di w. Verranno accettati dei pesi elevati solo nel caso in cui essi abbiano forte impatto positivo sulla loss function
Calcolando le derivate di $\tilde{J}$ rispetto a w e b, esse saranno uguali a:

$$\frac{\partial \tilde{J}}{\partial w} = \frac{\partial J}{\partial w} + \frac{\lambda}{n} w \tag{2.41}$$

$$\frac{\partial \tilde{J}}{\partial b} = \frac{\partial J}{\partial b} \tag{2.42}$$

Pertanto, la learning rule per i bias rimarrà invariata, ovvero:

$$b \to b' = b - \eta \cdot \frac{\partial J}{\partial b} \tag{2.43}$$

meentre per i pesi si avrà:

$$w \to w' = w - \eta \cdot \frac{\partial J}{\partial w} - \frac{\eta\lambda}{n} w = \left(1 - \frac{\eta\lambda}{n}\right) w - \eta \frac{\partial J}{\partial w} \tag{2.44}$$

In maniera analoga, è possibile vedere come varia l'aggiornamento di w e b nel caso del *mini-batch gradient descent*.

2.4.2 Regolarizzazione L_1

A differenza della regolarizzazione L_2, il termine aggiuntivo sarà uguale a:

$$\Omega = \frac{1}{n} \sum_x |w| \tag{2.45}$$

Anche la regolarizzazione L_1 risulta penalizzare i valori elevati w, tuttavia differisce dalla regolarizzazione L_2 in quanto tende a ridurre a 0 i pesi meno influenti. Pertanto, verranno considerati solamente i pesi che risultano avere un impatto maggiore al fine di migliorare il modello.

2.4.3 Dropout

Un'altra tecnica che si differenzia notevolmente dalle tecniche di regolarizzazione L_2 ed L_1 è il dropout, la quale è utilizzata solamente nelle reti neurali.
L'idea alla base del dropout consiste nel selezionare, ad ogni epoca, un numero casuale di nodi da 'eliminare momentaneamente' dalla rete, rimuovendone momentaneamente anche i pesi ad essi associati. All'epoca successiva, questi nodi verranno reinseriti all'interno della rete, mentre verranno momentaneamente eliminati ulteriori nodi con le rispettive connessioni.
Il vantaggio del dropout sta nel fatto che, da un punto di vista euristico, la cancellazione di diversi neuroni nel corso delle varie epoche equivale ad addestrare diverse reti neurali. La rete vera e propria assumerà le caratteristiche medie degli effetti calcolati per ogni rete privata di uno o più neuroni.

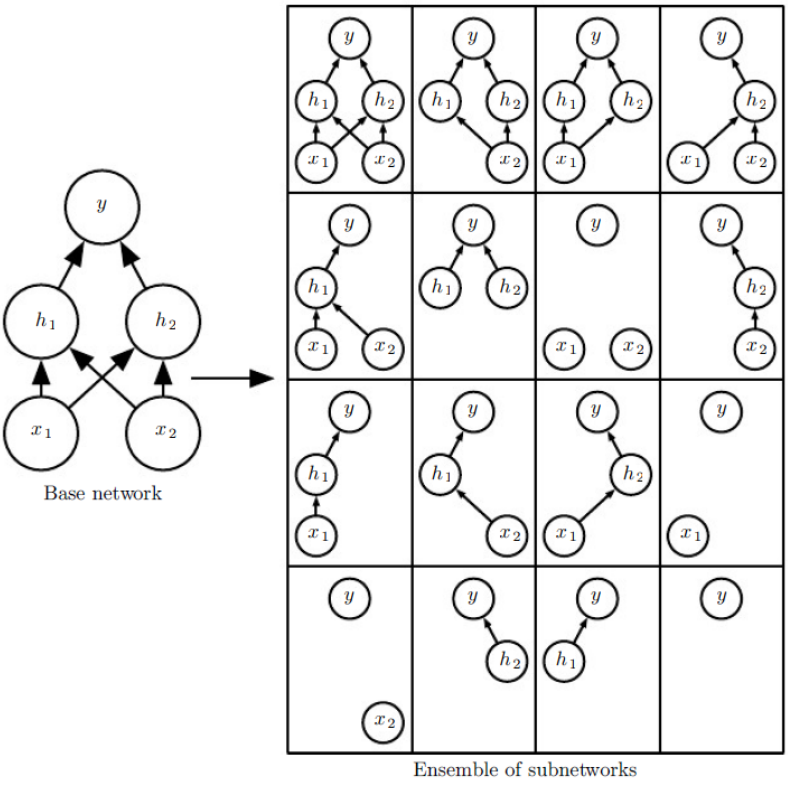


Figura 2.12: Esempio di rete neurale fully-connected avente due nodi di input, un hidden layer composto da due nodi ed un unico output. Di lato sono mostrate tutte le possibili sottoreti che possono essere costituite dall'eliminazione di uno o più nodi [4].

3 Reti Neurali Convoluzionali

3.1 Utilizzo e limiti delle reti neurali per l'elaborazione di immagini

Supponiamo di dover risolvere un problema di classificazione su un dataset di immagini. Le reti neurali profonde descritte in precedenza possono essere utilizzate per risolvere problemi di classificazione su immagini: un esempio applicativo consiste nella classificazione dalle immagini appartanenti al dataset MNIST [19]. In questo dataset sono presenti immagini in scala di grigi di dimensione 28×28 pixel raffiguranti un numero intero $\in \{0,1,...,9\}$ scritto a mano.

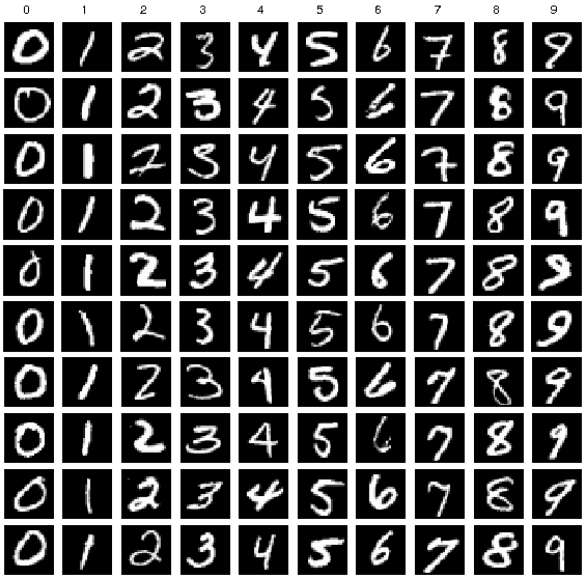


Figura 3.1: Esempio di immagini estratte dal dataset MNIST [20]

Un problema di classificazione sul dataset MNIST consiste nel classificare correttamente l'immagine data in input restituendo in output il numero ad esso associato.
Quando la rete neurale riceve in input un'immagine, essa viene interpretata come un array di pixel. Ciascun pixel assumerà un valore $\in [0,255]$ a seconda dell'intensità colore ad esso associato. Pertanto, un'immagine del dataset MNIST verrà vista come un array costituito da $28 \cdot 28 = 784$ elementi, ovvero le caratteristiche (altresì note come *features*) dell'immagine.

L'utilizzo di una semplice rete neurale *fully connected* per l'estrazione delle *features* di un'immagine presenta un problema significativo: per rappresentare un'immagine a colori (ovvero un'immagine in formato RGB) avente dimensione 8×8 pixel sono necessarie tre matrici di dimensione 8×8, ovvero una matrice per ciascun canale dell'immagine, rappresentabili come un'unica matrice avente dimensione $8 \times 8 \times 3$, nota anche come *tensore*, con ben 192 nodi nel primo layer. Il numero di nodi tende ad aumentare notevolmente con l'aumento delle dimensioni di un immagine: basti pensare che un'immagine a tre canali di dimensioni 256×256 pixel presenta ben $65.536 \cdot 3 = 196.608$ *features* solamente nel primo layer. Pertanto, collegare i nodi presenti nell'*input layer* con i nodi appartenenti al primo *hidden layer* sarà necessario un numero di pesi pari a $196.608 \cdot H_1$, con H_1 pari al numero di nodi presenti nel primo strato nascosto.

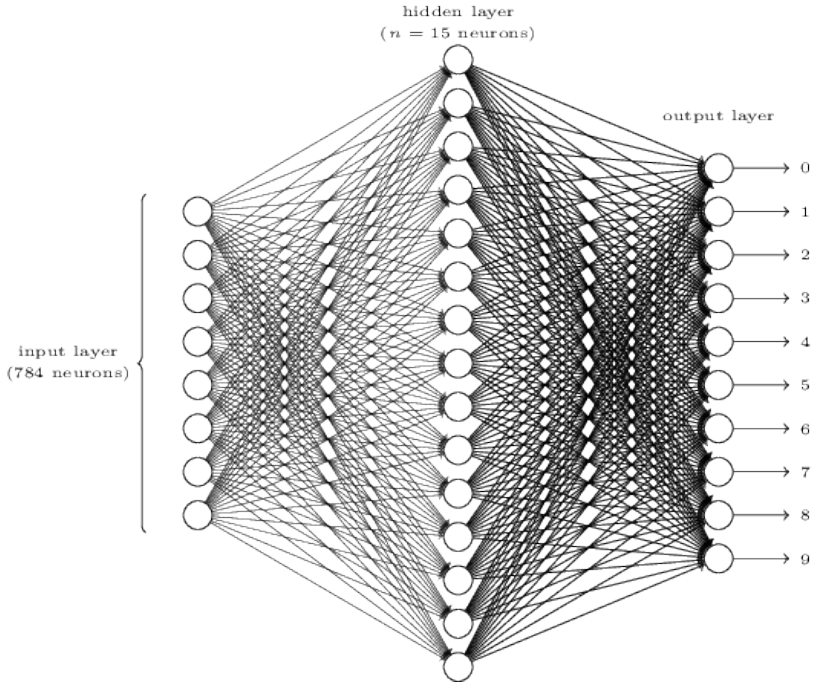


Figura 3.2: Esempio di rete neurale *fully connected* utilizzata per risolvere un problema di classificazione sul dataset MNIST [17]

Ma il problema principale risiede nel fatto che questo tipo di rete non risulta essere invariante rispetto a traslazioni e distorsioni dell'immagine [19].
Una soluzione al problema dell'elaborazione delle immagini è stata proposta nel 1998 da Yann LeCun [19], il quale propose un nuovo metodo di estrazione delle *features*: ogni immagine è suddivisa in diverse aree e da esse verranno estratte le caratteristiche (*features*) più significative, mediante l'utilizzo di *filtri*. Tale intuizione ha portato alla nascita delle reti neurali convoluzionali.
Le reti neurali convoluzionali, (*Convolutional Neural Networks*, *CNN*) sono una particolare tipologia di rete neurale utilizzate prevalentemente per l'elaborazione di immagini: la presenza del termine *convolutional* è dovuta al fatto che la rete utilizza al suo interno un'operazione matematica chiamata convoluzione. Gli strati chiamati *convolutional layers* vengono chiamati così in quanto presentano al loro interno operazioni di convoluzione in sostituzione del prodotto tra matrici [4].

3.1.1 Operatore di convoluzione

Come già detto in precedenza, la principale novità introdotta dalle *Convolutional Neural Networks* consiste nell'utilizzo di un operatore di convoluzione per l'estrazione delle *features*. Date due funzioni continue f e g, definiamo la convoluzione tra f e g, altresì indicata con $f * g$ il seguente integrale:

$$(f * g)(t) = \int f(\tau)g(t-\tau)d\tau \tag{3.1}$$

Nel caso in cui f e g siano due funzioni discrete, la convoluzione tra le due funzioni è così definita:

$$(f * g)(t) = \sum_{k=-\infty}^{+\infty} f(k)g(t-k) \tag{3.2}$$

Il risultato delle convoluzioni (3.2) e (3.1) può essere interpretato [21] come un mescolamento delle due funzioni ottenuto facendo scorrere la seconda funzione, detta anche *filtro*, sulla prima.
Inoltre, la convoluzione è commutativa, per cui si ha $f * g = g * f$.
In una CNN, la convoluzione avverrà tra un'immagine di cui vogliamo estrarre le caratteristiche ed un filtro. Poichè un'immagine è un array bidimensionale, ovvero una matrice, anche il filtro ad essa associata sarà una matrice, altresì nota come matrice di convoluzione o *kernel*. L'operatore di convoluzione, nel caso di immagini, sarà dunque uguale a:

$$(I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{3.3}$$

dove I e K saranno, rispettivamente, l'immagine ed il kernel (entrambi sotto forma di matrice).

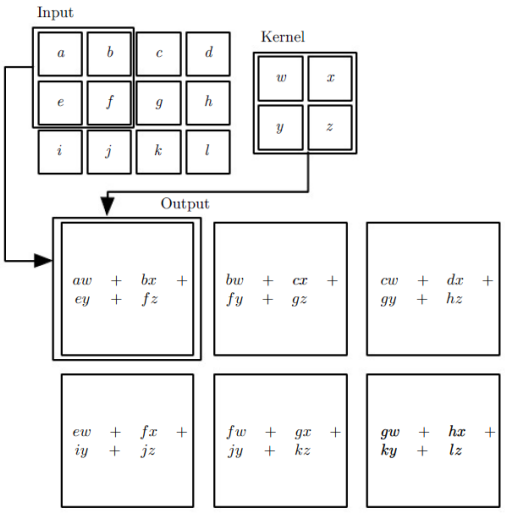


Figura 3.3: Esempio di convoluzione tra una matrice di input di dimensione 4×3 ed un kernel avente dimensione 2×2. Il risultato prodotto dalla convoluzione tra le due matrici sarà una matrice avente dimensione 3×2. [4]

3.2 Architettura di una CNN

Per costruire una CNN vengono utilizzati principalmente tre tipi di layer: il già citato *convolutional layer*, il *pooling layer* e il *fully-connected layer*, quest'ultimo che riprende i concetti spiegati all'interno del capitolo precedente.
Al fine di garantire un certo grado di invarianza rispetto a traslazioni e distorsioni, ciascuna rete neurale convoluzionale si basa su tre importanti concetti: il campo recettivo locale (*local receptive field*), pesi e bias condivisi (*shared weights and biases)* e la riduzione delle dimensioni delle *feature maps* mediante un'operazione detta *pooling* [19].

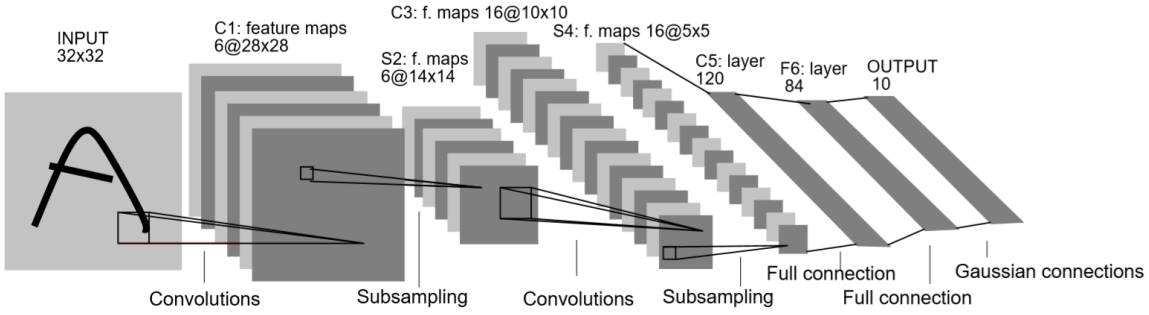


Figura 3.4: Esempio di architettura di una rete neurale convoluzionale (LeNet5) utilizzata per il riconoscimento di caratteri digitali [19].

Prima di parlare del *convolutional layer* è necessario introdurre il *local receptive field*:

- ***Local receptive field***: a differenza delle reti neurali *fully connected*, in cui gli input vengono rappresentati come componenti di un vettore, nelle CNN è possibile rappresentare ciascuna immagine di dimensione $m \times n$ come una matrice di nodi avente anch'essa dimensione $m \times n$. In questo caso, l'immagine di dimensione 28×28 verrà rappresentata come una matrice di neuroni avente anch'essa dimensione 28×28 in cui il neurone di posto (i, j) sarà associato all'intensità del pixel di posto (i, j) dell'immagine in input.
  A differenza delle reti neurali *fully connected*, non tutti i nodi dell'*input layer* verranno connessi a tutti i nodi del primo *hidden layer*: ciascun neurone appartenente al primo strato nascosto sarà collegato solamente ad una piccola regione dello strato di input, come mostrato in figura 3.5. Pertanto, la regione dell'*input layer* assicata al neurone del primo *hidden layer* costituirà il suo *local receptive field*.
  Per determinare i *local receptive field* associati agli ulteriori nodi del primo strato nascosto, è necessario spostare il primo campo recettivo di un passo k verso destra. Il passo con cui avviene lo shift è detto anche *stride*.
  In figura 3.6 è mostrato un esempio in cui il *local receptive field* associato al secondo nodo del primo strato nascosto è stato ottenuto mediante uno shift di passo 1 del primo campo recettivo preso in considerazione.

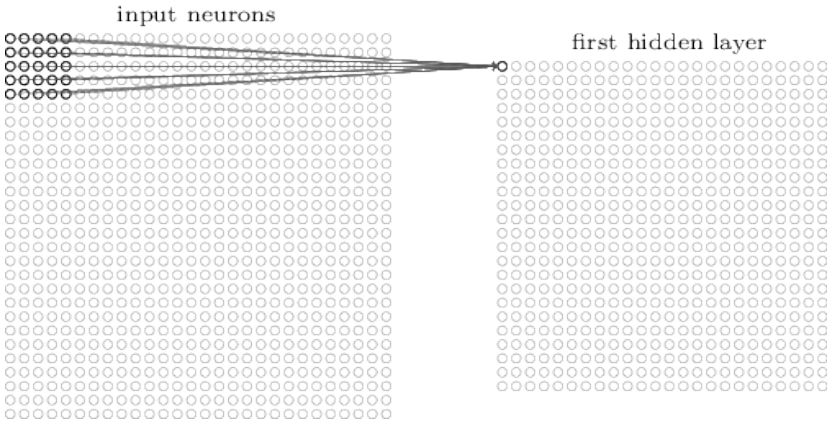


Figura 3.5: La figura mostra la connessione tra il primo nodo dell'hidden layer ed il *local receptive field* avente dimensione 5×5 ad esso associato [17].

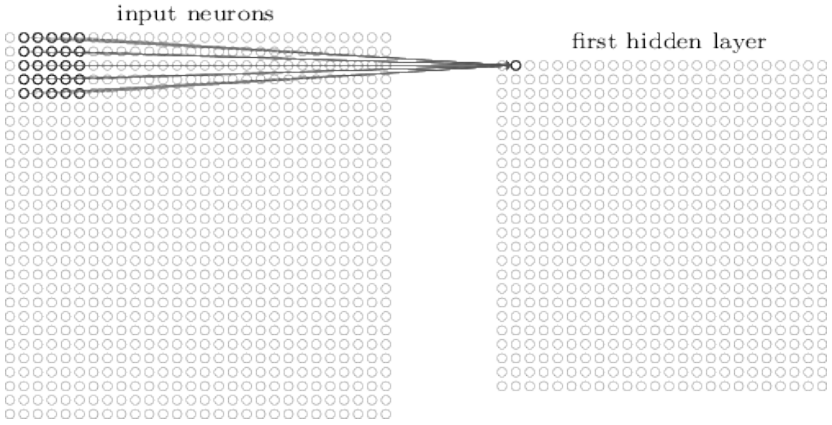


Figura 3.6: La figura mostra la connessione tra il secondo nodo dell'hidden layer ed il suo *local receptive field*, ottenuto mediante shift di *stride* 1 della prima regione. [17]

3.2.1 Convolutional Layer

Diamo un'occhiata alla struttura dello strato più importante di una CNN, ovvero lo strato convoluzionale (*convolutional layer*).
Introducendo il *local receptive field*, abbiamo visto come ciascun neurone del primo *hidden layer* 'vede' solamente alcuni neuroni appartenenti al primo strato.
E' importante analizzare bene come un *local receptive field* abbia un impatto sul valore del nodo del successivo strato: la matrice di nodi del campo ricettivo convolverà con una matrice di dimensione equivalente, detta anche *kernel*. Al risultato dell'operazione di convoluzione, espresso all'interno dell'equazione (3.3), corrisponderà il valore del nodo appartenente all'*hidden layer*.
Un layer avente al suo interno un'operazione di convoluzione, è detto appunto layer convoluzionale (*convolutional layer*).
Comee è mostrato all'interno delle figure 3.5 e 3.6, per determinare il valore del neurone di output è utilizzato sempre il medesimo filtro, il quale verrà condiviso per determinare il valore attribuito a ciascun neurone dell'*hidden layer*. Pertanto verranno utilizzati un numero di pesi pari alla dimensione del filtro (nel caso riportato all'interno della pagina precedente, è stato utilizzato un filtro di dimensione 5×5, per cui il numero di pesi è pari a $5 \cdot 5 = 25$), più l'aggiunta del bias. Se la rete fosse stata *fully connected*, a ciascun neurone dell'*hidden layer* sarebbero stati associati $28 \cdot 28$ pesi.
L'idea della condivisione di pesi è alla base del secondo concetto delle CNN:

- ***Shared weights and biases***: nell'esempio considerato nelle figure 3.5 e 3.6, ciascun *local receptive field* è composto da 25 nodi, mentre il primo strato nascosto avrà dimensione 24×24.
  L'output del nodo di posto (j, k) del primo strato nascosto sarà uguale a:

$$\sigma\left(b + \sum_{l=0}^{4}\sum_{m=0}^{4} w_{l,m} a_{j+l,k+m}\right) \tag{3.4}$$

  in cui indichiamo con σ la funzione di attivazione, mentre con $a_{x,y}$ denotiamo l'attivazione dell'input di posto (x, y).
  E' possibile notare che, come mostrato nella formula (3.4), ciascun neurone dello strato nascosto utilizza i medesimi pesi ed il medesimo bias: la condisione di w e b per tutti i neuroni dello strato convoluzionale evidenzia il fatto che ciascuno di esso apprende la medesima caratteristica, seppur in zone differenti dell'immagine. Quest'ultimo fatto evidenzia inoltre il fatto che, se una caratteristica localizzata in una particolare posizione dell'immagine viene rilevata, essa verrà rilevata anche se presente in differenti posizioni dell'immagine: ciò mette in risalto un'importante proprietà delle reti convoluzionali, ovvero l'invarianza rispetto alle traslazioni.
  Per questi motivi, la mappa generata partendo dall'*input layer* per mezzo di un filtro è detta anche mappa delle caratteristiche (*feature map*)

All'interno di un *convolutional layer*, spesso non è presente solamente un unico filtro ma ne vengono utilizzati molteplici: ciascuno di esso imparerà una caratteristica dell'immagine.

Inoltre, al fine di determinare la dimensione della *feature map* relativa al convolutional layer, bisogna prendere in considerazione tre iperparametri [22]:

1. *Profondità*: esso è uguale al numero di filtri utilizzati. Come detto in precedenza, ciascun filtro sarà utilizzato per imparare a localizzare differenti dettagli nell'immagine in input;

2. *Stride*: corrisponde al passo con cui facciamo scorrere il filtro e, di conseguenza, variare il *local receptive field* del layer da cui vogliamo estrarre informazioni. L'aumentare dello *stride* comporterà un valore più piccolo del volume di output;

3. *Zero-padding*: è una tecnica utilizzata per controllare la dimensione del volume di output. Essa consiste nell'aggiungere al volume in input un bordo di zeri. Questa tecnica assume una notevole importanza quando non si vuolee perdere determinate caratteristiche nel passaggio da uno strato ad un'altro nella rete.

La dimensione del volume in output sarà quindi dato dalla formula:

$$O = \frac{I - F + 2P}{S} + 1 \tag{3.5}$$

dove:

- I: dimensione del volume in input;
- F: dimensione del filtro;
- P: quantità di *zero-padding*;
- S: passo di *stride*.

Supponiamo che di avere un volume in input di dimensioni $W_0 \times H_0 \times D_0$. Riscrivendo l'equazione (3.5) per vedere come variano le tre dimensioni per il volume di output si avrà:

$$W_1 = \frac{W_0 - F + 2P}{S} + 1$$
$$H_1 = \frac{H_0 - F + 2P}{S} + 1$$
$$D_1 = N_F$$

E' opportuno scegliere dei valori di *stride* e *zero-padding* in modo che ogni dimensione del volume di output sia un numero intero.
La funzione (3.5) può essere applicata per determinare sia la larghezza sia l'altezza del volume di output. Indicando con N_F il numero di filtri presenti nello strato convoluzionale, il volume di output risultante avrà una profondità pari a N_F.
La figura 3.7 mostra come avviene la convoluzione tra un volume di input associato all'input layer avente dimensione $7 \times 7 \times 3$ e due filtri w_0 e w_1 aventi entrambi dimensione $3 \times 3 \times 3$.

In questo caso si avrà:

- $I = W \times H \times D = 7 \times 7 \times 3$
- $F = 3 \times 3 \times 3$
- $P = 1$
- $S = 2$
- $N_F = 2$

Pertanto, le dimensioni del volume di output saranno:

$$W = \frac{5 - 3 + 2 \cdot 1}{2} + 1 = 3$$

$$H = \frac{5 - 3 + 2 \cdot 1}{2} + 1 = 3$$

$$D = N_F = 2$$

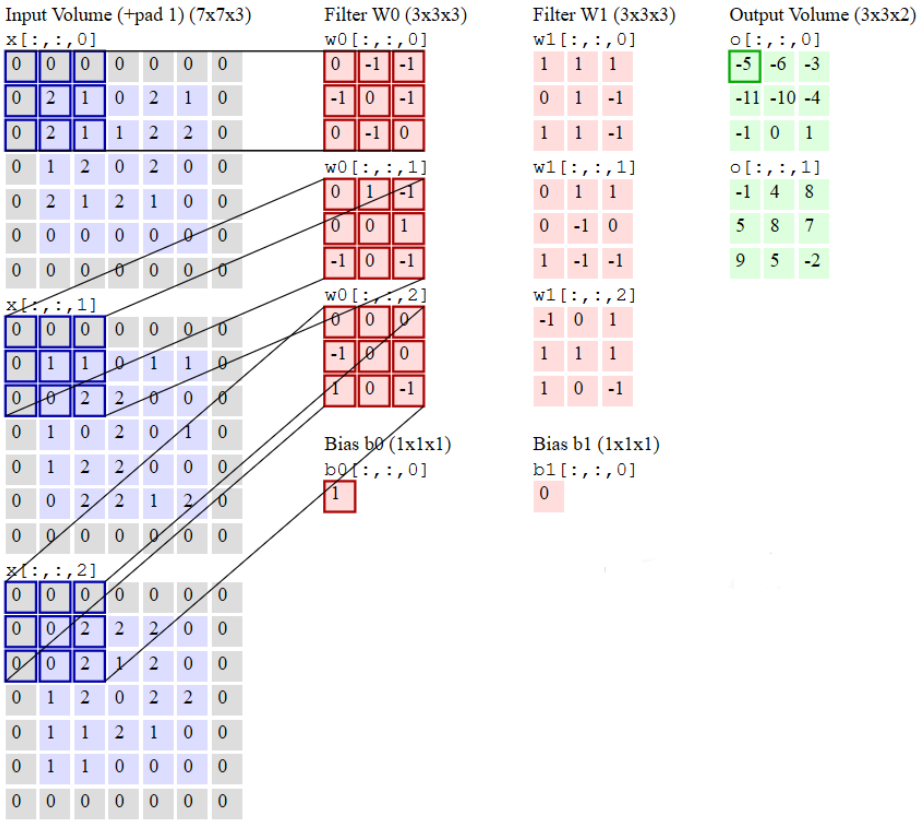


Figura 3.7: Esempio di convoluzione tra una matrice di dimensione $7 \times 7 \times 3$ e due filtri w_0 e w_1, con passo di stride pari a 2 [22].

Poichè la convoluzione restituisce una funzione lineare, una volta ottenuta la *feature map*, viene applicata la funzione d'attivazione reLU, la cui funzionalità sarà quella di porre uguali a 0 tutti i nodi aventi valori negativi.
Spesso una rete neurale convoluzionale possiede molteplici strati convoluzionali: partendo dagli *input layers*, i primi strati convoluzionali estraggono caratteristiche di basso livello dell'immagine (come ad esempio angoli, linee orizzontali e verticali). Viceversa, gli ultimi layer convoluzionali permettono di estrarre caratteristiche significative dell'immagine, quali volti o oggetti.

3.2.2 Pooling Layer

Lo scopo dei *pooling layers* è quello di ridurre le dimensionalità dell'immagine.
Il concetto alla base del *pooling* è molto semplice: ogni matrixe di pixel verrà suddivisa in diversi sottogruppi e, per ognuno di essi, vengono sostituiti da una loro statistica sommaria, in modo da ottenere una matrice avente dimensioni ridotte.
Bisogna tuttavia tener conto del fatto che seppur l'operazione di pooling porta alla riduzione della lunghezza e della larghezza della matrice, la profondità rimarrà inalterata. Supponiamo di dover sottoporre ad un'operazione di pooling una matrice avente dimensioni $W_1 \times H_1 \times D_1$. Gli iperametri necessari per l'operazione di pooling sono il passo di stride S e la dimensione del filtro F. Il volume di output risultante avrà dimensione pari a:

$$W_2 = \frac{W_1 - F}{S} - 1$$

$$H_2 = \frac{H_2 - F}{S} - 1$$

$$D_2 = D_1$$

Vengono comunemente utilizzate due diverse tecniche di pooling: *max pooling* e *average pooling*. Vediamo come opera ciascuna di essa:

Max Pooling

E' la tecnica di *pooling* maggiormente utilizzata.

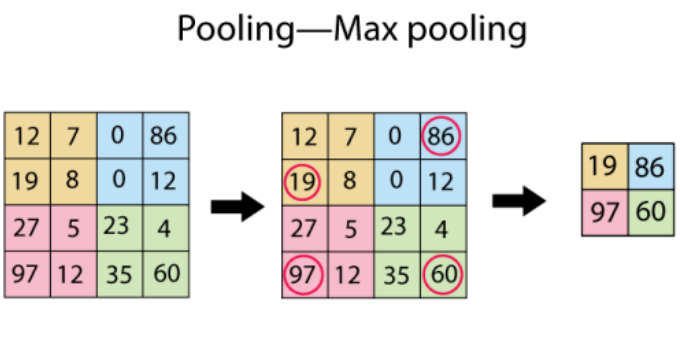


Figura 3.8: Applicazione del *max pooling* su una matrice di dimensione 4×4.[23]

Nell'esempio, il *max pooling* opera utilizzando un filtro di dimensione 2×2 ed uno *stride* di 2×2, per ogni sottoinsieme di pixel, ne verrà estratto solamente il pixel avente valore massimo. Prendendo in riferimento il primo blocco, il quale è composto dai valori [12,7,19,8], applicando il *max pooling* verrà restituito solamente il valore massimo presente all'interno del blocco, cioè 19.

Average Pooling

A differenza del *max pooling*, la riduzione dei pixel non avverrà selezionando il massimo valore presente in ogni blocco.

Considerando tutti i valori presenti all'interno di ciascun blocco, la riduzione della

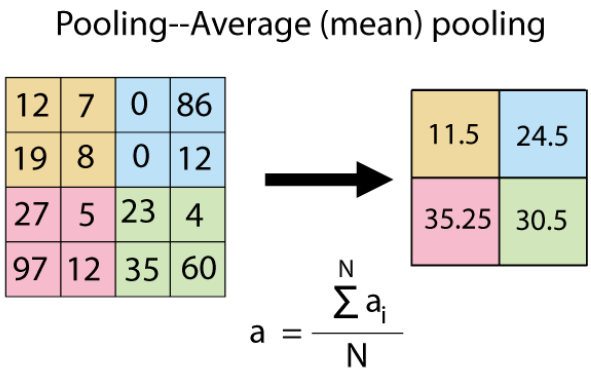


Figura 3.9: Applicazione dell'*average pooling* su una matrice di dimensione 4×4.[23]

dimensione porterà alla scelta del valore ottenuto facendo la media tra tutti i valori presenti all'interno del blocco. Ad esempio, prendendo in considerazione i valori relativi al primo blocco, ovvero [12,7,19,8], l'*average pooling* restituirà la media tra essi, ovvero $(12 + 7 + 19 + 8)/4 = 11.5$, e così via per tutti i blocchi della matrice.

3.2.3 Fully-Connected Layer

Una volta terminate tutte le operazioni che coinvolgono gli strati convoluzionali e di pooling, si otterrà una matrice di dimensione $W \times H \times D$.
Mediante un'operazione chiamata *flattening*, è possibile rappresentare una matrice avente dimensioni $W \times H \times D$ all'interno di un vettore di dimensione $W \cdot H \cdot D$.
A questo punto è possibile Il funzionamento dei *fully-connected layers*.
La struttura di tutti gli strati *fully-connected* sarà la medesima descritta all'interno del Capitolo 2. L'ultimo layer *fully-connected*, il quale sarà uguale all'*output layer* avrà un numero di nodi pari al numero totale di classi definite per il problema di classificazionee.
Al vettore in output, generato mediante funzione di classificazione softmax, corrisponderà la probabilità associata dalla rete che l'oggetto appartenga alla classe i-esima.

3.3 Data Augmentation

Il miglior modo per aumentare la capacità del modello ad adattarsi a nuovi dati consiste nell'aumentare le dimensioni del dataset. Tuttavia, non sempre è possibile avere a disposizione dataset di grandi dimensioni. Una soluzione a questo problema consiste nel manipolare i dati presenti aggiungendo particolari effetti al fine di fornire contesti diversi alla rete affinchè essa possa estrapolarne meglio il contenuto semantico.
L'aggiunta di una o più alterazioni ai dati originali viene detta *data augmentation*: essa si presta molto bene per problemi di classificazione di immagini.
Le pratiche di *data augmentation* più comuni consistono nell'aumentare dati modificando la geometria delle immagini, quali riflessione (*Flip*), rotazione, traslazione, e la modifica di contrasto e/o luminosità.
Poichè l'aggiunta di nuovi dettagli significativi alle immagini può portare il modello ad apprendere meglio determinate caratteristiche, è molto importante individuare una pipeline di *data augmentation* che riesca ad adattarsi ai dati di training e a mantenere un'ottima capacità di generalizzazione.

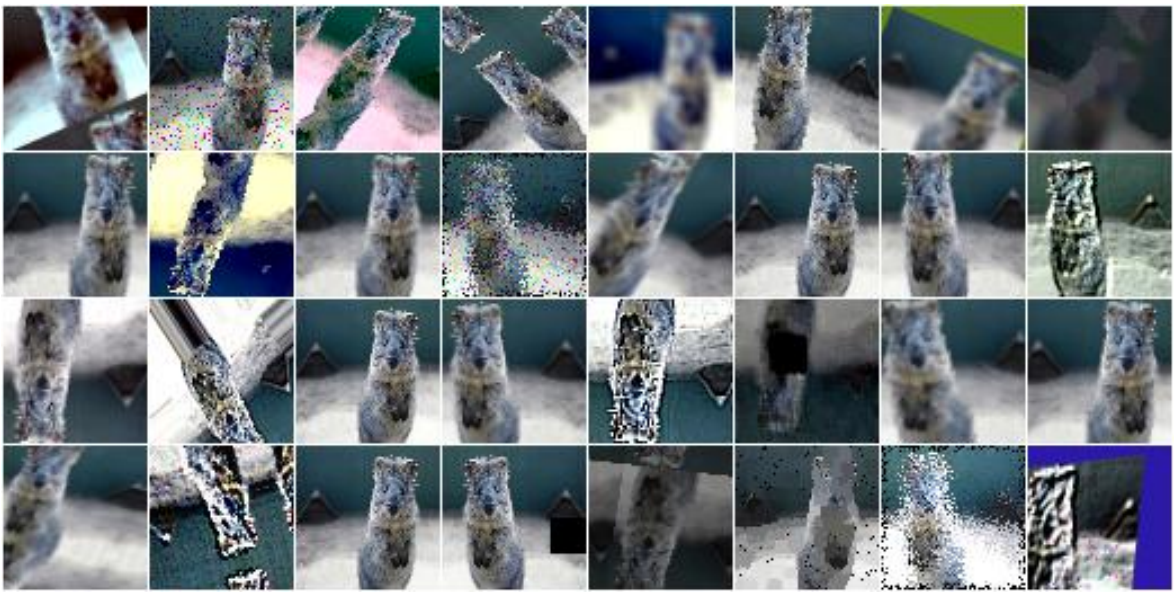


Figura 3.10: Effetti di data augmentation applicati ad una singola immagine [24].

Un aspetto molto importante da tener conto nella scelta di una o più tecniche di data augmentation da applicare ad un'immagine è il tipo di problema di classificazione che si sta affrontando. Supponiamo di avere a che fare con un problema di classificazione di loghi e di dover applicare un effetto di data augmentation che modifichi il colore di un'immagine: è abbastanza evidente che, poichè un logo possiede determinate caratteristiche, tra cui appunto il colore, questo tipo di effetto produrrà esempi di classificazione errati.
Un ulteriore esempio di cattiva scelta di una pipeline di data augmentation riguarda un problema di classificazione di caratteri: se applichiamo un effetto di rotazione di 180° ad un'immagine raffigurante il carattere 'b', esso verrà rappresentato come una 'q', pertanto condizionerà negativamente le capacità di classificazione del modello.

4 Mask R-CNN

Una rete neurale convoluzionale può essere utilizzata per molteplici obiettivi: da problemi di classificazione fino a problemi di segmentazione semantica delle istanze. Effettuare una segmentazione semantica significa dividere un'immagine in diversi insiemi di pixel che devono essere opportunamente etichettati e classificati.
Quando è necessario effettuare la segmentazione semantica di ogni singola istanza (*semantic instance segmentation* o, più brevemente indicata come *instance segmentation*), l'obiettivo diventa quello di differenziare ogni singolo oggetto opportunamente identificato (*object detection*) e classificato al fine di determinarne l'esatta posizione e differenziarlo da qualsiasi altro oggetto (anche appartenente alla medesima classe). Nel 2014

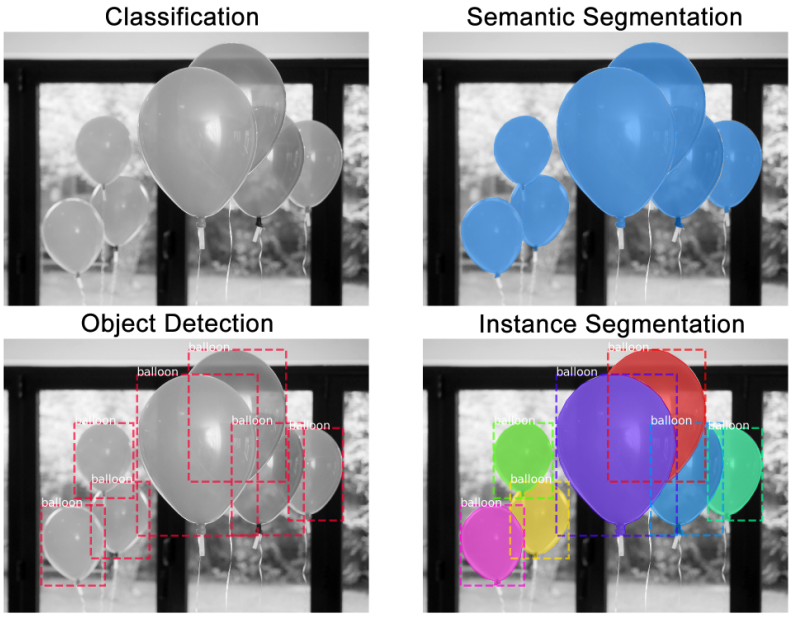


Figura 4.1: Variazione dell'output di un'immagine al variare del problema: classificazione (in alto a sinistra), *object detection* (in basso a sinistra), *semantic segmentation* (in alto a destra) e *instance segmentation* (in basso a destra) [25].

fu proposta una rete neurale, la *Region-based Convolutional Neural Network* (R-CNN) che permetteva di risolvere problemi di *object detection*. Nel corso degli anni R-CNN venne migliorata notevolmente, presentando Fast R-CNN e Faster R-CNN, le quali hanno superato, in termini di performance e accuratezza la precedente rete neurale.
L'evoluzione di Faster R-CNN, Mask R-CNN, permette di risolvere problemi di segmentazione di singole istanze ed è la rete neurale utilizzata nell'applicazione descritta all'interno del Capitolo 5.

4.1 Reti neurali Region-based CNN

Al fine di comprendere il funzionamento di Mask R-CNN, è opportuno evidenziare gli aspetti principali delle reti *region-based* CNN.
Le reti *region-based* che precedono Mask R-CNN sono state proposte per risolvere problemi di *object detection*, ovvero un'attività che richiede la classificazione e il rilevamento di tutti gli oggetti presenii in ciascuna immagine [26].
Utilizzeremo il termine 'riconoscimento di oggetto' in senso ampio per comprendere sia la classificazione delle immagini (un'attività che richiede un algoritmo per determinare quali classi di oggetti sono presenti nell'immagine) sia il rilevamento di oggetti (un'attività che richiede un algoritmo per localizzare tutti gli oggetti presenti nell'immagine.

4.1.1 R-CNN

La rete R-CNN [27], *Region-based Convolutional Neural Network*, proposta nel 2014, trasforma un problema di *object detection* in un problema di classificazione.
Il funzionamento è il seguente: data un'immagine, vengono innanzitutto estratte circa 2000 possibili regioni d'interesse (*RoI*) mediante utilizzo dell' algoritmo *Selective Search* [28], successivamente verranno estratte le caratteristiche di ogni singola regione utilizzando una CNN ed infine, verranno classificate le regioni sulla base delle caratteristiche estratte applicando una SVM per la classificazione ed una regressione lineare al fine di restringere la bounding box dell'oggetto.

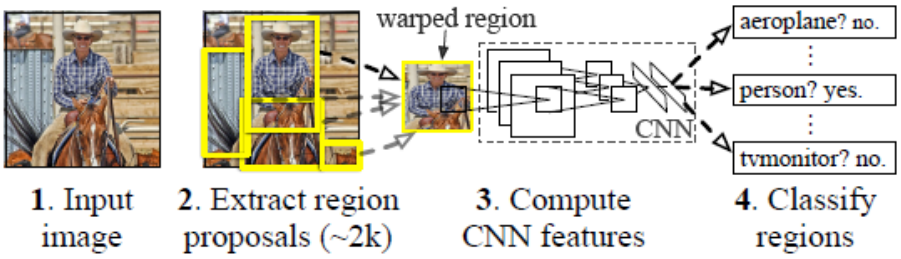


Figura 4.2: Illustrazione del funzionamento di R-CNN [27]

Il problema principale di R-CNN è dovuto al costo computazionale elevato: ogni regione proposta di un'immagine deve essere successivamente classificata da una rete convoluzionale e quest'operazione risulta essere molto dispendiosa, in quanto dovrà essere ripetuta per ogni singola regione estratta. Inoltre, ogni regressore delle bounding box verrà allenato singolarmente, senza condivisione di informazioni. Inoltre, R-CNN non utilizza una singola SVM, bensì un numero equivalente al numero di classi: in fase di training risulta quindi necessario allenare singolarmente ciascuna SVM.

4.1.2 Fast R-CNN

Il problema relativo alla lentezza della rete è stato risolto l'anno successivo (2015) [29]. In R-CNN abbiamo visto che vengono innanzitutto estratte le possibili regioni d'interesse e successivamente classificate da una CNN. In Fast R-CNN le due operazioni

vengono "invertite": l'estrazione delle regioni d'interesse non è la prima operazione effettuata dalla rete, bensì avviene sull'ultima mappa delle caratteristiche estratte dalla rete. In Fast R-CNN, infatti, i primi strati convoluzionali hanno lo scopo di evidenziare le caratteristiche più importanti dell'immagine e facilitare il passo successivo di selezione delle *features* al fine di determinare le regioni d'interesse. Inoltre, la classificazione non viene più effettuata da una SVM per classe, bensì viene utilizzato un singolo classificatore softmax. Fast R-CNN ha migliorato in maniera significativa le performance

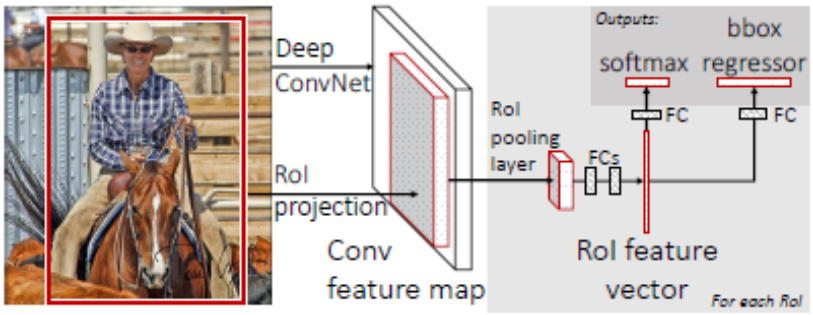


Figura 4.3: Architettura di Fast R-CNN [29]

ottenute da R-CNN sia in termini di velocità sia in termine di accuratezza, tuttavia non ha risolto completamente il problema della selezione delle possibili regioni d'interesse: l'algoritmo di *Selective Search* risulta essere ancora lento ed estrae nuovamente nuovamente un numero eccessivo ed inutile di regioni d'interesse.
Fast R-CNN possiede due differenti output layers: il primo restituisce una distribuzione di probabilità discreta $p = (p_0, .., p_K)$ su $K + 1$ categorie, in cui p è calcolato mediante una funzione softmax sui $K + 1$ output del *fully-connected layer*; il secondo restituisce le coordinate $t^k = (t^k_x, t^k_y, t^k_w, t^k_h)$ delle bounding box rifinite dal regressore.
La loss function associata ad ogni RoI estratta sarà dunque uguale a:

$$\mathcal{L}(p, u, t^u, v) = \mathcal{L}_{cls}(p.u) + \lambda[u \geq 1]\mathcal{L}_{box}(t^u, v) \tag{4.1}$$

La loss function relativa al classificatore sarà uguale a:

$$\mathcal{L}_{cls}(p, u) = -\log p_u \tag{4.2}$$

dove p_u è la probabilità che l'oggetto appartenga alla classe u.
La loss function relativa al regressore sarà invece uguale a:

$$\mathcal{L}_{box}(t^u, v) = \sum_{i \in \{x,y,w,h\}} smooth_{L_1}(t^u_i - v_i) \tag{4.3}$$

in cui $smooth_{L1}$ sarà uguale a:

$$smooth_{L_1}(x) = \begin{cases} 0.5x^2 & |x| < 1 \\ |x| - 0.5 & |x| \geq 1 \end{cases} \tag{4.4}$$

e t^u sarà uguale alla bounding box predetta per la classe u.
L'impatto dell'iperparametro λ utilizzato all'interno dell'equazione (4.1) regolerà l'impatto di $\mathcal{L}_{cls}$ e $\mathcal{L}_{box}$ ai fini del calcolo della loss function $\mathcal{L}$: maggiore sarà il valore di

$\mathcal{L}$, maggiore sarà il contributo dato da $\mathcal{L}_{box}$ rispetto a $\mathcal{L}_{cls}$.
Inoltre, poichè a ciascuna classe è associato un *class_id* (il cui valore di default associato alla classe *background* è pari ad 1), con la terminologia $[u \geq 1]$ si fa riferimento alle sole classi non classificate come *background* dalla rete.

4.1.3 Faster R-CNN

L'idea base di Faster R-CNN [30] consiste nel migliorare le performance di Fast R-CNN modificando il processo di selezione delle regioni d'interesse. Poichè le precedenti versioni utilizzavano un algoritmo di selective search esterno, e dunque non facente parte della rete neurale vera e propria, esso non poteva essere allenato in fase di training. In Fast R-CNN i primi strati convoluzionali avevano lo scopo di estrarre la mappa delle caratteristiche per permettere l'estrazione delle regioni d'interesse: Faster R-CNN aggiunge ulteriori strati che permettono l'estrazione delle regioni d'interesse mediante una rete convoluzionale vera e propria, *Region Proposal Network* (RPN), la quale sostituisce definitivamente l'algoritmo *Selective Search*.
A livello implementativo, è possiblile dunque vedere Faster R-CNN come la composizione di RPN e Fast R-CNN.

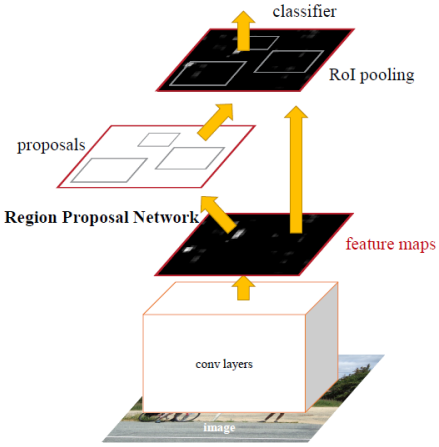


Figura 4.4: Architettura di Faster R-CNN [30]

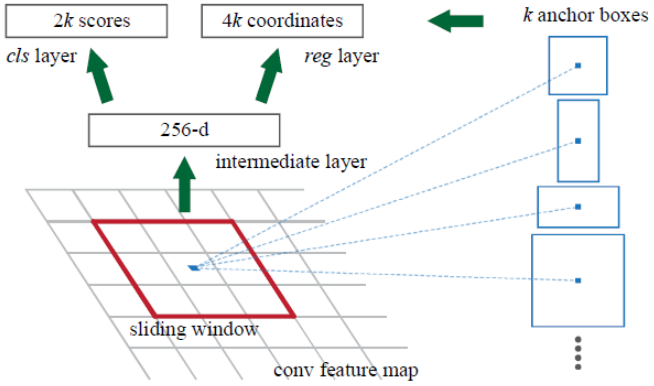


Figura 4.5: Region Proposal Network [30]

Region Proposal Network

La rete *Region Proposal Network* (RPN) prende un'immagine in input e restituisce in output un insieme di regioni proposte: a ciascuna di queste regioni è associata la probabilità che l'oggetto si trovi effettivamente in tale area. Con il termine 'regione' la rete intende un'area avente dimensione rettangolare in cui è possibile rilevare un oggetto.
Una volta arrivati all'ultimo strato della prima CNN presente all'interno di Faster R-CNN, una finestra scorrevole avente dimensione $n \times n$ ($n = 3$, [30]) viene fatta scorrere lungo la *feature map* al fine di determinare i riquadri delle *region proposal*, denotati anche come ancore (*anchors*). Ogni volta che viene fatta scorrere la finestra lungo la *feature map*, vengono individuate al più k ancore al variare di differenti valori di scala e *aspect ratio*.
Di default si hanno 3 differenti valori di scala e 3 differenti *aspect ratio*, per un totale

di $k = 3 \cdot 3 = 9$ ancore per ciascuna finestra scorrevole. Per ciascuna di queste regioni verrà associata la probabilità di contenere un'istanza (*score* e le coordinatee dell'*anchor box* individuato.
Pertanto, per una *feature map* avente dimensione $W \times H$ verranno estratte $W \cdot H \cdot k$ *anchor boxes*.
Una volta determinate le *region proposal*, esse verranno nuovamente elaborate dalla rete: gli strati successivi saranno equivalenti a quelli proposti da Fast R-CNN, per cui sarà presente un pooling layer, un classsificatore softmax ed un regressore delle bounding box.
La loss function relativa ad ogni RoI estratta sarà uguale a:

$$\mathcal{L} = \mathcal{L}_{cls} + \mathcal{L}_{box} \tag{4.5}$$

Essa si differenzia dalla loss function utilizzata in Fast R-CNN per l'aggiunta dei termini di normalizzazione. Riscriviamo dunque il tutto come:

$$\mathcal{L}(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i \mathcal{L}_{cls}(p_i, p_i^*) + \frac{\lambda}{N_{box}} \sum_i p_i^* \cdot smooth_{L_1}(t_i - t_i^*) \tag{4.6}$$

in cui $\mathcal{L}_{cls}$ sarà la log-loss calcolata su due classi (si può facilmente passare da un problema di classificazione multiclasse ad un problema di classificazione binario considerando la probabilità che l'i-esima ancora appartenga ad una determinata classe contro la probabilità che non vi appartenga):

$$\mathcal{L}_{cls}(p_i, p_i^*) = -p_i^* \log p_i - (1 - p_i^*) \log(1 - p_i) \tag{4.7}$$

mentre la definizione di $\mathcal{L}_{box}$ sarà la medesima utilizzata da Fast R-CNN.
All'interno di (4.6), si è utilizzata la seguente notazione[31]:

- p_i: probabilità predetta che l'i-esima ancora appartenga ad un oggetto;
- p_i^*: probabilità ground truth che l'i-esima ancora sia un oggetto;
- t_i: coordinate predette;
- t_i^*: coordinate ground truth;
- N_{cls}, N_{box} termini di normalizzazione relativi, rispettivamente, a $\mathcal{L}_{cls}$ e a $\mathcal{L}_{box}$;
- λ parametro che bilancia $\mathcal{L}_{cls}$ e $\mathcal{L}_{box}$

4.2 Deep Residual Networks

Uno dei problemi delle reti neurali profonde è costituito dal fatto che, all'aumentare della profondità della rete, l'accuratezza del modello diminuisce. Questo problema non è causato dall'overfitting, bensì è dovuto all'aumento dell'errore di training [32].
Un'idea innovativa è stata proposta nel 2015 mediante l'introduzione delle cosiddette reti neurali residuali (*residual neural networks*) [33]. Nelle CNN finora analizzate, dato un input x il quale verrà sottoposto a differenti sequenze di operazioni all'interno dei layer successivi, il suo valore in output sarà denotato con la notazione $F(x)$. Il layer successivo assumerà in input il valore restituito dal precedente output, ovvero $F(x)$, e così via.
In una rete neurale residuale, l'output non sarà più uguale a $F(x)$, bensì assumerà il valore del suo residuo, ovvero $H(x) = F(x) + x$. Un blocco di operazioni di questo tipo è detto anche *building block*.

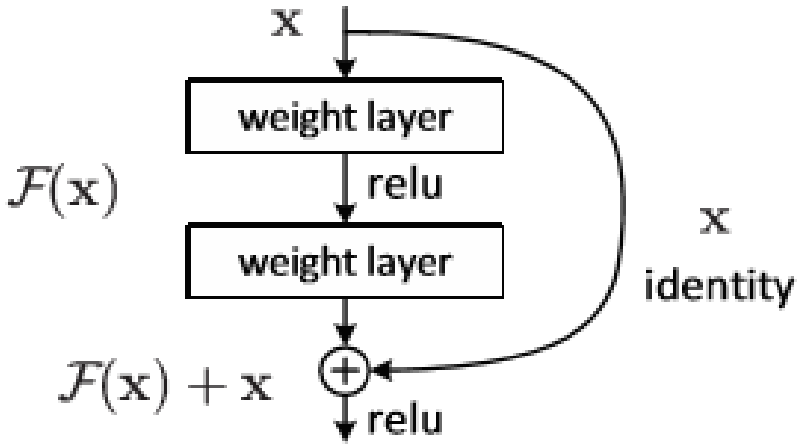


Figura 4.6: Esempio di building block [33]

I risultati ottenuti mediante l'utilizzo delle reti neurali residuali hanno permesso l'implementazione di reti le quali, nonostante l'elevata profondità, non riscontrano problemi di accuratezza dovuto all'elevato numero di layer. Per rendere ancora meglio l'idea, prima dell'introduzione delle reti neurali residuali, il modello di rete più profondo avesse un numero di layer pari a 22 [34], mentre oggi sono presenti reti neurali residuali (*ResNet*) aventi oltre 1000 layer [35].

4.3 Mask R-CNN

Mask R-CNN [36], utilizzato per problemi di *instance segmentation*, è un'estensione di Faster R-CNN: rispetto a quest'ultima, vi è l'aggiunta di un terzo branch che permette la previsione della maschera di un'istanza. Quest'operazione avviene in parallelo con i due branch già presenti all'interno di Faster R-CNN (il regressore della bounding box ed il classificatore).Nello specfico, per ogni *Region Of Interest* (RoI), Mask R-CNN genererà una maschera avente dimensione 28×28, la quale verrà successivamente espansa fino ad adattarsi alle dimensioni della bounding box corrispondente.
Per ogni istanza individuata, Mask R-CNN restituirà in output la classe d'appartenenza, la sua bounding box e una maschera binaria ad essa sovrapposta.

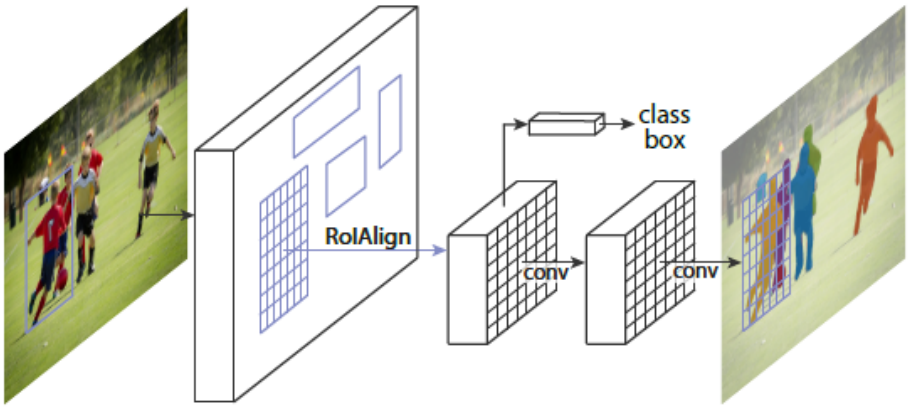


Figura 4.7: Architettura di Mask R-CNN [36].

4.3.1 Backbone

All'interno di [36] sono state valutate differenti versioni di backbone da utilizzare per l'estrazione delle *features* (ovvero il blocco denotato come CNN all'interno della figura 4.8), quali ResNet e ResNeXt, entrambe aventi 50 o 101 layers e la cui estrazione delle *features* avviene a partire dall'ultimo layer convoluzionale appartenente al quarto o al quinto blocco (ad esempio, un backbone ResNet-101-C4 indicherà una ResNet avente profondità pari a 101 layer e la cui estrazione delle features avviene a partire dall'ultimo layer convoluzionale appartenente al quarto blocco).
Per ottenere *features* ancora più accurate è stata proposta una seconda rete convoluzionale da inserire al termine del primo backbone: la *Feature Pyramid Network* (FPN) [37]. Quest'ultima sfrutta una struttura piramidale per l'estrazione di diverse *features* aventi differenti scale. E' stato evidenziato come l'approccio combinato di ResNet e FPN abbia permesso un eccellente guadagno in termini di precisione e velocità di estrazione delle *features* [36].

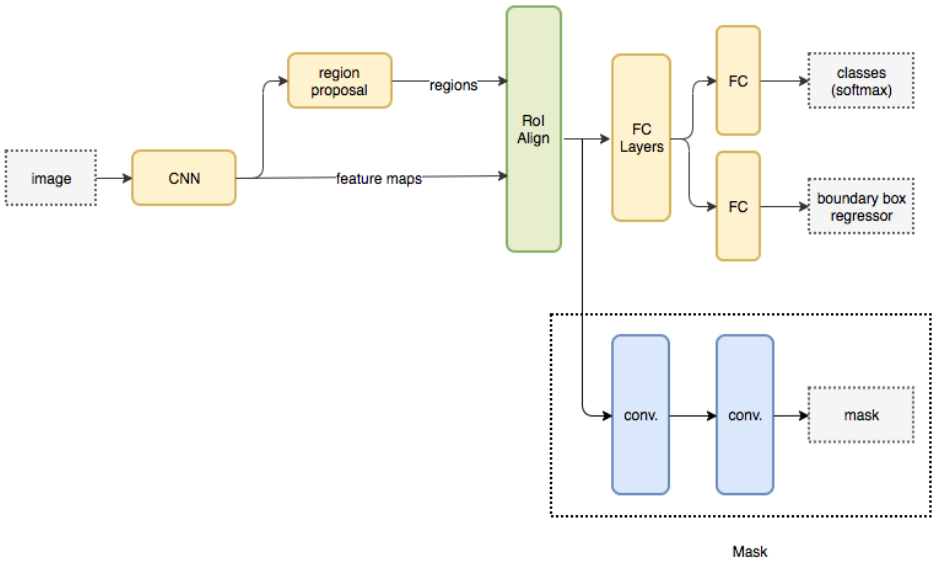


Figura 4.8: Illustrazione degli layer presenti all'interno di Mask R-CNN [38].

4.3.2 Loss Function

La loss function relativa ad ogni RoI estratta sarà uguale a:

$$\mathcal{L} = \mathcal{L}_{cls} + \mathcal{L}_{box} + \mathcal{L}_{mask} \tag{4.8}$$

dove $\mathcal{L}_{cls}$ e $\mathcal{L}_{box}$ saranno uguali alle loss function relative al classificatore ed al regressore delle bounding box utilizzate da Faster R-CNN e da Fast R-CNN.
Relativamente al calcolo di $\mathcal{L}_{mask}$, a ciascuna RoI viene associata una sola maschera *ground truth* e verrò applicata una funzione d'attivazione sigmoide ad ogni pixel della maschera. Il branch associato alla previsione della maschera genererà maschere binarie aventi dimensioni $m \times m$ per ciascuna delle K possibili classi. Pertanto, in totale si genereranno $K \cdot m^2$ possibili maschere, ciascuna associata ad una diversa classe.
$\mathcal{L}_{mask}$ viene definita come la media tra le binary cross-entropy loss function, in cui è inclusa la k-esima maschera se la regione è associata alla k-esima maschera *ground truth*:

$$\mathcal{L}_{mask} = -\frac{1}{m^2} \sum_{1 \leq i,j \leq m} [y_{ij} \log \hat{y}_{ij}^k + (1 - y_{ij}) \log(1 - \hat{y}_{ij}^k)] \tag{4.9}$$

4.3.3 RoI Align

Un'importante novità introdotta da Mask R-CNN è l'utilizzo del RoI Align in sostituzione del RoI Pooling per l'estrazione delle RoI di una *feature map*.
Supponiamo di avere un'immagine di dimensione 128×128 ed una *feature map* ad essa associata avente dimensioni 25×25. Vogliamo estrarre una RoI di dimensione 15×15 dell'immagine originale.
Pertanto, sarebbe necessario estrarre una regione di pixel dalla *feature map* pari a $m \times m$, con $m = \frac{25*15}{128} \approx 2.93$.

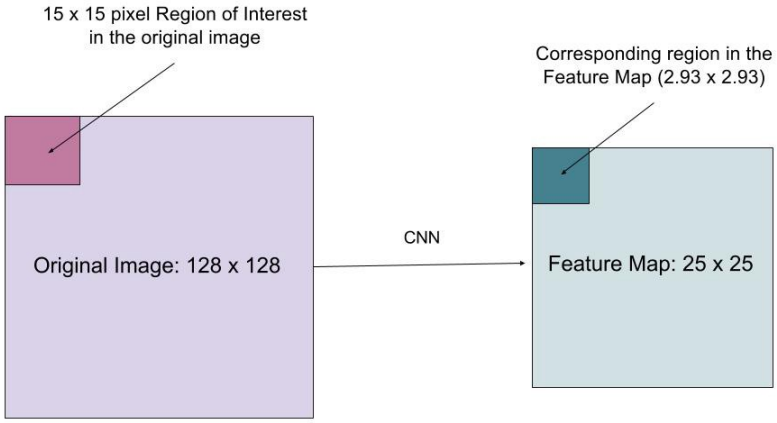


Figura 4.9: L'esempio mostra come viene mappata una RoI da un'immmagine ad una feature map [39].

Utilizzando il RoI Pooling, non sarà possibile estrarre una regione di dimensioni 2.93×2.93, bensì verrà considerata una regione approssimata pari a 2×2 (ovvero verrà considerata la parte intera di m). Queest'approssimazione produce una perdita di 0.93 pixel per dimensione: ad essa è associata una perdita di informazioni in quanto, le tecniche di pooling non terranno conto dei pixel perduti.
Questo problema di disallineamento dei pixel è stato risolto grazie al RoI Align layer, il quale non approssima più la dimensione delle regioni estratte. Considerando nuovamente l'esempio proposto, la regione estratta avrà effettivamente una dimensione 2.93×2.93.

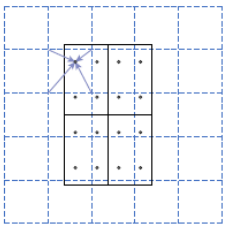


Figura 4.10: In figura viene mostrata una *feature map* in cui avviene l'estrazione di una RoI di dimensione 2×2, ciascuno avente quattro diversi punti di campionamento. Il valore dei punti di campionamento verrà calcolato mediante interpolazione bilineare.

4.4 Metriche di valutazione delle performance di Mask R-CNN

4.4.1 Intersection over Union

E' una metrica utilizzata per quantificare la percentuale di sovrapposizione tra la maschera reale e la maschera predetta in output dal modello. Essa è il rapporto tra l'area d'intersezione e l'area di unione delle due bounding box.

$$IoU = \frac{A \cap B}{A \cup B} \tag{4.10}$$

Figura 4.11: Esempio di calcolo della IoU per diverse bounding box [40].

Relativamente al calcolo della IoU di due maschere, essa misura il numero di pixel in comune tra le due maschere diviso il numero totale di pixel presenti in entrambe le maschere.

4.4.2 Precision-Recall

Oltre a determinare bounding box e maschera di ogni singola istanza, *Mask R-CNN* deve saper classificare correttamente l'istanza individuata.
Più in generale, in un problema di classificazione, è necessario stabilire se un input x avente output y venga classificato correttamente dal modello. A seconda della classe che verrà restituita in output, si potrà determinare se il dato in input sia stato classificato correttamente.
E' necessario distinguere i seguenti casi possibili:

- *True Positive (TP)*: è un esempio che viene classificato correttamente dal modello come appartenente ad una classe;
- *True Negative (TN)*: è un esempio che viene classificato correttamente dal modello come non appartenente ad una classe;
- *False Positive (FP)*: è un esempio che viene classsificato in maniera non corretta dal modello come appartenente ad una classe;
- *False Negative (FN)*: è un esempio che viene classificato in maniera incorretta dal modello come non appartenente ad una classe.

E' possibile quindi definire la *precision* come:

$$\frac{TP}{TP + FP} \tag{4.11}$$

la quale rappresenta la percentuale di positivi classificati correttamente dal modello. Analogamente, si definisce la *recall* come:

$$\frac{TP}{TP + FN} \tag{4.12}$$

Essa rappresenta, tra tutti gli elementi appartenenti ad una certa classe, la percentuale di positivi classificati correttamente.

4.4.3 Average Precision

L'utilizzo di *precision* e *recall* non è sufficiente a stabilire la bontà di un classificatore. Ciò è dovuto al fatto che all'aumentare di una delle due metriche corrisponde il decremento dell'altra, e viceversa.
Pertanto, per rappresentare opportunamente la capacità di classificazione del modello, le due metriche vengono valutate nella cosiddetta curva *precision-recall*, la quale avrà nelle ascisse i valori relativi alla *recall* e nelle ordinate i valori relativi alla *precision*. L'area sottesa dalla curva precision-recall è detta *average precision* (AP) ed assumerà un valore compreso tra 0 ed 1. Essa sarà dunque uguale a:

$$AP = \int_0^1 p(r)dr \tag{4.13}$$

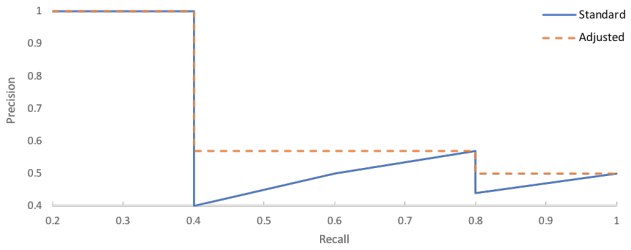


Figura 4.12: Esempio di curva *precision-recall* in cui l'aria sottesa evidenzia l'average precision.[41]

La media dei valori di *average precision* ottenuti per classe costituirà la *mean average precision* (mAP).
Per semplificare il calcolo dell'integrale, si procede al calcolo di AP preferendone una versione approssimata mediante interpolazione. Nel corso degli anni [42] è cambiata la metodologia di calcolo di AP: nel 2008, la Pascal VOC2008 Challenge propose di calcolarla come media dei valori massimi di precision calcolati ad 11 livelli standard di recall (0,0.1,0.2, ...,1.0), ovvero:

$$AP = \frac{1}{11}(AP_r(0) + AP_r(0.1) + ... + AP_r(1)) \tag{4.14}$$

cioè

$$AP = \frac{1}{11}\sum_{r=0}^{11} AP(r) = \\ = \frac{1}{11}\sum_{r=0}^{11} p_{interp}(r) \tag{4.15}$$

dove

$$p_{interp}(r) = \max_{\tilde{r}\geq r} p(\tilde{r}) \tag{4.16}$$

Tuttavia, questa definizione di AP risultava essere poco precisa, oltre a perdere la capacità di valutare modelli aventi bassi valori di AP.
Una successiva definizione di AP fu proposta sempre da Pascal VOC qualche anno dopo [42], e consisteva nel calcolare $p(r_i)$ non più come (4.15), bensì ogni volta in cui il suo valore decresce.
La definizione di AP proposta fu la seguente:

$$AP = \sum (r_{n+1} - r_n) p_{interp}(r_{n+1}) \tag{4.17}$$

dove

$$p_{interp}(r_{n+1}) = \max_{\tilde{r}>r_{n+1}} p(\tilde{r}) \tag{4.18}$$

4.4.4 Metriche di COCO

L'equazione (4.17) è quella utilizzata anche da MS-COCO per il calcolo delle proprie metriche di valutazione del modello.
Un dataset COCO viene valutato su 12 differenti metriche:

```
Average Precision (AP):
   AP                  % AP at IoU=.50:.05:.95 (primary challenge metric)
   AP^IoU=.50          % AP at IoU=.50 (PASCAL VOC metric)
   AP^IoU=.75          % AP at IoU=.75 (strict metric)
AP Across Scales:
   AP^small            % AP for small objects: area < 32^2
   AP^medium           % AP for medium objects: 32^2 < area < 96^2
   AP^large            % AP for large objects: area > 96^2
Average Recall (AR):
   AR^max=1            % AR given 1 detection per image
   AR^max=10           % AR given 10 detections per image
   AR^max=100          % AR given 100 detections per image
AR Across Scales:
   AR^small            % AR for small objects: area < 32^2
   AR^medium           % AR for medium objects: 32^2 < area < 96^2
   AR^large            % AR for large objects: area > 96^2
```

Figura 4.13: Metriche utilizzate per valutare un dataset in formato COCO. Immagine estratta da *http://cocodataset.org/#detection-eval*.

Nella notazione in figura 4.13, con AP e AR si intendono i valori di AP e AR mediati su ciascuna classe: non vi è dunque distinzione di notazione tra AP ed mAP e tra AR e mAR (all'interno di questo paragrafo tali notazioni saranno equivalenti).

$AP^{IoU=.50}$ e $AP^{IoU=.75}$ sono le mAP calcolate rispettivamente prendendo in considerazione solamente le istanze aventi IoU, rispettivamente, maggiore o uguale a 0.50 e 0.75. La metrica più importante da considerare per valutare la performance del modello è AP calcolata a IoU=.50 : .05 : .95: essa calcola le AP facendo variare il valore di IoU (IoU = 0.50 significa che verranno considerate solamente le istanze aventi IoU$\geq$ 0.50), ovvero incrementandolo di 0.05 ad ogni valutazione completa del dataset , per poi riportarne la media tra esse. Le metriche di *AP Across Scales* sono invece calcolate prendendo in considerazione le AP calcolate considerando la dimensione delle istanze rilevate. Nello specifico si avrà:

- AP^{small}: è l' Average Precision calcolata tenendo in considerazione solamente istanze aventi un'area $< 32^2$;
- AP^{medium}: è l' Average Precision calcolata tenendo in considerazione solamente istanze aventi un'area compresa tra 32^2 e 64^2;
- AP^{large}: è l'Average Precision calcolata tenendo in considerazioe solamente istanze aventi un'area maggiore di 64^2.

in cui per area si intende il numero di pixel all'interno della segmentation mask. Definiamo l'*Average Recall* come la media calcolata su tutte le categorie e i valori di IoU del valore massimo di recall ottenuto da un numero fissato di *detections* per immagine. Pertanto $AR^{max=1}$ sarà uguale alla media calcolata su tutte le categorie e sui valori di IoU del valore di recall ottenuto prendendo in considerazione al più un'istanza per immagine. Il medesimo ragionamento ci porterà alla definizione di $AR^{max=10}$ e $AR^{max=100}$.
Infine, ai valori di *AR Across Scales* corrisponderà la media sulle categorie e sui valori di IoU del massimo valore di recall ottenuto prendendo in considerazione solamente istanze aventi area $< 32^2$ (AR^{small}), $32^2 <$ area $< 64^2$ (AR^{medium}) e area $> 64^2$ (AR^{large}).

5 Un'applicazione: Retail Segmentation

L'obiettivo della parte sperimentale consiste in un'applicazione di Mask R-CNN volta al riconoscimento e alla segmentazione di oggetti di uso quotidiano destinati alla vendita. L'implementazione di Mask R-CNN utilizzata da cui si è partiti per la creazione e successiva valutazione del modello è la versione presente nel repository [43], scritta in linguaggio Python, per poi successivamente riadattare il progetto originale al problema di *instance segmentation* proposto.
Poichè l'addestramento della rete partendo da pesi casuali sarebbe stato dispendioso e avrebbe portato difficilmente a risultati soddisfacenti nel breve periodo, sono stati sfruttati i pesi già addestrati, effettuando un'operazione di *transfer learning*: essa consiste nell'effettuare l'addestramento partendo da pesi già allenati.
In fase di training è stata utilizzata una GPU NVIDIA GeForce GTX 1080 8Gb.
La rete neurale è stata addestrata al fine di riconoscere e segmentare opportunamente oggetti appartenenti alle seguenti quattro classi:

1. Bottiglia (*Bottle*);
2. Scatola (*Box*);
3. Barattolo (*Jar*);
4. Pacchetto (*Packet*).

5.0.1 Descrizione del dataset

Il dataset è composto da 435 immagini appartenenti al training set, 60 immagini appartenenti al validation set e 60 immagini appartenenti al test set.

	Training Set	***Validation Set***	***Test Set***
Bottle	679	102	59
Box	647	91	57
Jar	664	88	57
Packet	653	102	60
Total	2.643	383	233

Per effettuare la segmentazione manuale di ogni singola istanza da inserire all'interno del dataset è stato utilizzato il tool presente in *https://www.labelbox.com/*
Nel corso del lavoro, il training set è stato spesso ampliato, al fine di ottenere il maggior numero possibile di istanze per ogni singola classe. Il numero di immagini presenti all'interno del validation set è stato mantenuto costante lungo tutto il processo di training per permettere di confrontare le validation loss avendo sempre lo stesso banco di prova.

5.1 Training del modello

La parte più lunga e più significativa del lavoro è senza dubbio la fase di training del modello. Mask R-CNN possiede numerosi parametri la cui variazione ha un impatto più o meno considerevole ai fini del risultato finale.
Un modello ottenuto è ritenuto accettabile se esso possiede una buona capacità di apprendere correttamente le istanze presenti all'interno del training set mantenendo una buona capacità di generalizzazione delle conoscenze apprese. Ci si aspetta, ovviamente, che all'aumentare delle epoche, le loss functionrelative al training set tendano a diminuire. Per ottenere un modello soddisfacente, è necessario che la loss function relativa al validation set (validation loss) non cresca all'aumentare delle epoche: qualora si verifichi un incremento notevole della validation loss, significa che il modello sta andando incontro al problema dell'overfitting: esso sta imparando talmente bene gli esempi presenti nel training set da non riuscire a performare in maniera efficiente in un contesto più generico. Quando si presentano queste problematiche, è necessario determinarne le cause, le quali possono essere molteplici:

- **Learning rate** troppo alto/basso;
- La **policy di learning rate decay** ha poco impatto sull'apprendimento;
- **Weight decay** troppo alto/basso;
- La pipeline di **data augmentation** scelta causa l'aggiunta di immagini poco significative (gli effetti aggiunti alle nuove immagini modificano poco le immagini originali). Viceversa, effetti di data augmentation troppo complessi possono complicare notevolmente l'apprendimento;
- E' necessario modificare il **backbone**;
- E' necessario modificare le dimensioni dell'**image shape**, la quale rappresenta la dimensione dell'immagine riscalata da *Mask R-CNN*;
- E' opportuno modificare i pesi solo di determinati layer della rete (**heads/all**).

Inizialmente sono state utilizzate le configurazioni standard utilizzate in [43]. Di seguito sono indicati i parametri maggiormente significativi:

Hyperparameter	*Value*
Backbone	resnet101
Batch size	1
Image shape	[1024 1024 3]
Learning rate	0.001
Learning rate decay	none
Data augmentation	none
Steps per epoch	500
Validation steps	50
Weight decay	0.0001
Weights trained	All

La prima fase di training è stata dedicata principalmente allo studio dell'impatto degli iperparametri di Mask R-CNN sul modello finale. Ciascun modello ottenuto in questa fase è stato ottenuto dopo 30 epoche di training, ciascuna avente 500 steps per epoca. Inoltre, le caratteristiche della GPU utilizzata non hanno permmesso di estendere il batch size ad un numero maggiore di 1.
Innanzitutto, al fine di velocizzare il processo di training, è stato modificato *image shape* impostandone il valore a [512 512 3]. Tale modifica ha permesso di dimezzare il tempo necessario per il training del modello senza avere un impatto negativo sulle performance della rete in termini di variazione di loss/validation loss function.
Sono state valutate le performance ottenute modificando il *backbone* della rete. Tuttavia, l'utilizzo di *resnet50* al posto di *resnet101*, nonostante abbia aumentato la velocità di training, non ha migliorato l'apprendimento poichè la validation loss tendeva a crescere con maggiore pendenza.
Poichè sono stati utilizzati dei pesi già addestrati in precedenza, è stato necessario valutare a più ripresa se convenisse o meno effettuare il training del modello aggiornando solamente i pesi relativi agli ultimi layer della rete, ovvero i *fully-connected layers*, e 'congelando' i pesi relativi agli ulteriori layer. Per evidenziare quali layer verranno coinvolti nel processo di training, verrà utilizzata la terminologia *heads* (per indicare l'aggiornamento dei pesi relativi ai soli fully-connected layers) e *all* (per indicare l'aggiornameento di tutti i pesi della rete). Spesso sono state provate configurazioni ibride, ad esempio:

- variazione del learning rate ogni x steps;
- variazione dei layer coinvolti nel processo di aggiornamento dei pesi: ad esempio, su n epoche di training, nelle prime $n/2$ epoche verranno aggiornati solamente i pesi delle *heads* e nelle successive $n/2$ tutti i pesi della rete verranno aggiornati;
- variazione del learning rate ogni x steps + variazioni dei layer coinvolti nel processo di aggiornamento dei pesi.

Un problema ricorrente in fase di training è stato quello relativo all'overfitting: partendo da un dataset composto da 120 immagini, è stato riscontrato come, all'aumentare delle epoche, i valori di validation loss non tendevano a decrescere/rimanere costanti, bensì tendeva a crescere all'aumentare delle epoche. Pertanto è stato necessario aumentare il numero di immagini presenti all'interno del dataset, al fine di ottenere un modello che tenda a non overfittare all'aumentare del numero di epoche.

Number of images	120	172	194	359	402	435
Bottle	204	272	408	604	668	679
Box	359	440	440	585	612	647
Jar	228	304	398	577	657	664
Packet	239	414	415	582	615	653
Total	1.030	1.430	1.661	2.348	2.552	2.643

Tabella 5.1: La tabella rappresenta la variazione del numero di istanze ad ogni incremento di immagini all'interno del dataset

Grazie a *TensorBoard* è stato possibile monitorare l'andamento del processo di training osservando come variano le loss functions relative al training set e al validation set all'aumentare delle epoche.

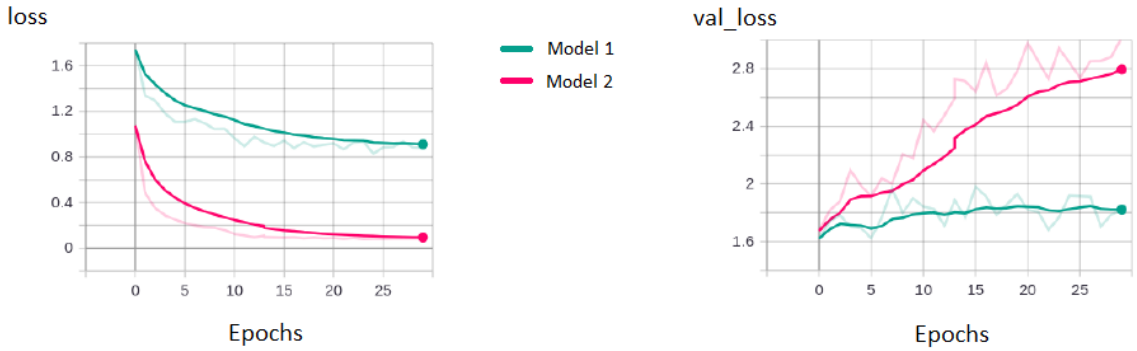


Figura 5.1: Confronto tra due differenti configurazioni di training. L'esempio è volto a mostrare un confronto tra un modello soggetto al problema di overfitting ed un altro che, nonostante abbia appreso molto meno in fase di training rispetto al precedente, possiede una discreta capacità di generalizzazione.

Analizzando l'andamento del training dei due modelli riportati in figura 5.1 è possibile osservare che al termine delle 30 epoche, nonostante il modello 1 abbia dei valori relativi alla loss function più elevati rispetto al modello 2, esso risulta performare meglio in termini di validation loss: il modello 2 tende ad andare in overfitting (al crescere delle epoche, la validation loss ad essa associata tende a crescere), a differenza del primo che mantiene, all'aumentare del numero di epoche, valori di validation loss più o meno costante.
Una volta individuate la configurazioni degli iperparametri aventi le performance migliori in termini di decremento della loss function e di non incremento della validation loss function, si è provveduto ad analizzare i training ottenuti su un numero di epoche

più elevato, per analizzarne l'andamento sul lungo periodo. Analizzando il comportamento delle due funzioni su 30 epoche, si è potuto evincere che valori troppo bassi di *weight decay* (impostato inizialmente ad un valore di default di 10^{-4} e successivamente analizzato per valori compresi tra 10^{-1} e 10^{-5}) portano il modello ad avere scarsa capacità di generalizzazione e, di conseguenza, ad andare incontro al problema dell'overfitting.
E' possibile fare le medesime considerazioni per la scelta del *learning rate*: se esso viene settato ad un valore $< 10^{-4}$, l'apprendimento procederà in maniera molto lenta e, seppur la validation loss ottenuta con valori di *learning rate* molto bassi, tende generalmente a crescere poco e/o a decrescere, l'apprendimento è stato spesso poco soddisfacente.
Un passo molto importante al fine di determinare la configurazione ottimale in fase di training riguarda la scelta della pipeline di data augmentation. Per tale scopo, sono state effettuate diverse prove con pipeline più o meno complesse, oltre alla valutazione del modello senza alcun effetto di data augmentation. Come era lecito aspettarsi, l'aggiunta di "falsi dati" ha permesso un aumento sostanziale delle performance. La scelta di una configurazione di data augmentation volta a garantire performance acceettabili sul medio/lungo periodo ha costituito la fase più lunga nel processo di training.
Di seguito vengono elencate alcune tra le pipeline di data augmentation utilizzate in fase di training del modello:

```
augmentation =  iaa.Fliplr(0.5)
```

Figura 5.2: Data augmentation - Pipeline 1

E' la più semplice pipeline di data augmentation utilizzata: ad ogni immagine del training set potrà essere applicato un effetto di riflessione orizzonale (*Flip left/right*), con una probabilità 50%.
La possibile aggiunta di un singolo effetto potrebbe non migliorare di molto le performance del modello, pertanto sono state provate configurazioni differenti:

```
augmentation = iaa.SomeOf((0,2), [
iaa.Fliplr(0.5),
iaa.Flipud(0.5)
])
```

Figura 5.3: Data augmentation - Pipeline 2

Rispetto alla prima pipeline descritta, a ciascuna immagine (se selezionata casualmente per l'applicazione degli effetti di data augmentation) potrà essere applicato uno o due effetti di data augmentation scelti tra:

- Riflessione orizzontale
- Riflessione verticale (*Flip up/down*)

Ciascun effetto potrà essere applicato con una probabilità del 50%.

```
augmentation = iaa.SomeOf((0, 3), [
  iaa.Fliplr(0.5),
  iaa.Flipud(0.5),
  iaa.OneOf([iaa.Affine(rotate=60),
  iaa.Affine(rotate=120),
  iaa.Affine(rotate=180)],
  ),
  iaa.Affine(scale={"x": (0.8, 1.2), "y": (0.8, 1.2)}),
  iaa.GaussianBlur(sigma=(0.0, 5.0))
])
```

Figura 5.4: Data augmentation - Pipeline 3

```
augmentation = iaa.SomeOf((0, 3), [
  iaa.Fliplr(0.5),
  iaa.Flipud(0.5),
  iaa.OneOf([iaa.Affine(rotate=60),
  iaa.Affine(rotate=120),
  iaa.Affine(rotate=180)],
  ),
  iaa.Affine(scale={"x": (0.8, 1.2), "y": (0.8, 1.2)}),
  iaa.AverageBlur(k=(2, 11))
])
```

Figura 5.5: Data augmentation - Pipeline 4

Aggiungere solamente effetti di riflessione all'immagine potrebbe non essere sufficiente ad aggiungere dettagli significativi ai dati in grado di migliorare le performance della rete. Non sempre un'immagine o un video può avere un'alta risoluzione, pertanto sono state provate due configurazioni di data augmentation (il cui codice è riportato nelle figure 5.4 e 5.5) le quali applicano fino ad un massimo di tre effetti scelto tra le seguenti opzioni:

- Riflessione orizzontale dell'immagine
- Riflessione verticale dell'immagine
- Un effetto scelto casualmente tra:
  - Rotazione di 60°
  - Rotazione di 120°
  - Rotazione di 180°
- Distorsione dell'immagine
- Sfocatura

Queste due configurazioni differiscono solamente per l'effetto di sfocatura dell'immagine: nel primo caso sono stati testati gli effetti di sfocatura ottenuti mediante *Gaussian Blur*, nel secondo caso è stata utilizzata la tecnica di *Average Blur*.

E' stato possibile verificare che, mantenendo invariati i valori degli ulteriori iperparametri, la pipeline in cui è presente l'effetto di *Gaussian Blur* produce modelli tendenti in maniera minore al problema dell'overfitting rispetto ai modelli ottenute con la pipeline in cui è presente l'effetto di *Average Blur*.

```
augmentation = iaa.SomeOf((0, 5), [
  iaa.Fliplr(0.5),
  iaa.Flipud(0.5),
  iaa.OneOf([iaa.Affine(rotate=60),
  iaa.Affine(rotate=120),
  iaa.Affine(rotate=180)],
  ),
  iaa.Affine(scale={"x": (0.8, 1.2), "y": (0.8, 1.2)}),
  iaa.ContrastNormalization((0.5, 1.5), per_channel=0.5),
  iaa.GaussianBlur(sigma=(0.0, 5.0))
])
```

Figura 5.6: Data augmentation - Pipeline 5

Quest'ultima pipeline differisce dalle precedenti per il numero massimo di effetti applicabili all'immagine (5, contro i 3 delle precedenti ultime due configurazioni descritte nelle pipeline 3 e 4). Inoltre, tra gli effetti applicabili, vi è l'aggiunta del contrasto normalizzato.

Al termine dell'analisi dei modelli ottenuti dopo 30 epoche, si è potuto osservare che le migliori performance sono state ottenute con un valore di weight decay pari a 0.05 e per valori iniziali di learning rate pari a 10^{-3}, mentre le pipeline di data augmentation che ha ottenuto i risultati migliori è stata la pipeline 5, mentre altri risultati ritenuti accettabili su 30 epoche sono stati ottenuti utilizzando la pipeline 2. Dato il numero di epoche ridotto in fase di training, non è stato possibile valutare se la scelta di una policy di learning rate decay e la scelta di allenare solamente i pesi relativi alle *heads* potessero influire positivamente sull'andamento del risultato finale del modello.
Aumentando il numero di epoche di training fino a 270, è stato possibile valutare quali tra le migliori configurazioni degli iperparametri ottenute inialmente contnuassero ad avere un buon impatto sul modello anche sul lungo periodo.
Come era lecito aspettarsi, non tutte le configurazioni ottimali ottenute su 30 epoche hanno ottenuto performance accettabili anche su 270 epoche.
Sulla base dei risultati ottenuti analizzando l'andamento dei training sul lungo periodo, è stato possibile arrivare alle seguenti conclusioni:

- L'aggiornamento dei pesi relativo alle sole *heads* (*fully connected layers*) comporta valori accettabili di validation loss, tuttavia i valori di loss function risultano molto più alti rispetto a configurazioni che prevedevano le modifiche di tutti i pesi della rete o di configurazioni ibride. In merito, sono stati ottenuti discreti risultati modificando solamente i pesi dei *fully-connected layers* per 135 epoche per poi effettuare l'aggiornamento su tutti i pesi della rete per le successive epoche di training.
- Applicando una policy di learning rate decay ogni x steps non sono stati ottenuti modelli che performassero bene in termini di valutazione delle due loss functions.

- E' stato necessario aumentare nuovamente il numero di immagini presenti all'interno del training set.

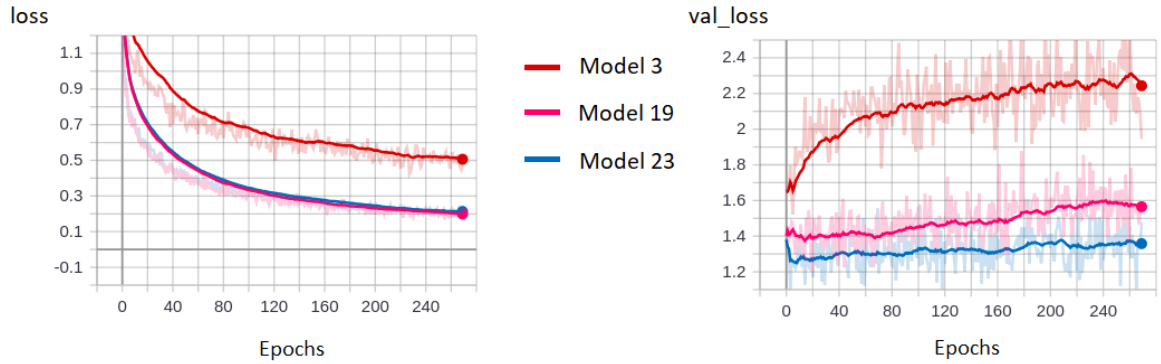


Figura 5.7: Confronto tra tre differenti modelli ottenuti con le medesime configurazioni di iperparametri al variare del numero di immagini presenti nel training set.

Nella figura 5.7 è possibile notare che, a parità di configurazioni degli iperparametri, il numero di immagini presenti nel training set ha migliorato notevolmente i risultati finali in termini di decremento della loss function/non aumento della validation loss function: il primo modello (i cui andamenti delle loss sono indicati in marrone) è stato ottenuto con un training set di 265 immagini, mentre il modello migliore (evidenziato in blu) è stato allenato su 402 immagini. Di seguito sono indicate le configurazioni di iperparametri utilizzati per determinare i migliori modelli ottenuti relativamente alla valutazione dell'andamento delle loss function relative al training e al validation set:

Hyperparameter	*Value*
Backbone	resnet101
Batch size	1
Image shape	[512 512 3]
Learning rate	0.001
Learning rate decay	none
Data augmentation	Pipeline 5
Steps per epoch	500
Validation steps	50
Weight decay	0.05
Weights trained	All
Number of images	359

Tabella 5.2: Modello 19

Hyperparameter	*Value*
Backbone	resnet101
Batch size	1
Image shape	[512 512 3]
Learning rate	0.001
Learning rate decay	none
Data augmentation	Pipeline 5
Steps per epoch	500
Validation steps	50
Weight decay	0.05
Weights trained	All
Number of images	402

Tabella 5.3: Modello 23

Hyperparameter	*Value*
Backbone	resnet101
Batch size	1
Image shape	[512 512 3]
Learning rate	0.001
Learning rate decay	none
Data augmentation	Pipeline 5
Steps per epoch	500
Validation steps	50
Weight decay	0.05
Weights trained	*Heads* (1-30) *All*(others)
Number of images	402

Tabella 5.4: Modello 25

Hyperparameter	*Value*
Backbone	resnet101
Batch size	1
Image shape	[512 512 3]
Learning rate	0.001
Learning rate decay	none
Data augmentation	Pipeline 5
Steps per epoch	500
Validation steps	50
Weight decay	0.05
Weights trained	All
Number of images	435

Tabella 5.5: Modello 26

5.2 Analisi delle performance dei modelli ottenuti

La sola valutazione dell'andamento grafico delle due loss functions non è sufficiente per concludere che il modello A sia migliore del modello B. E' stato quindi necessario valutare i migliori modelli scelti sulla base dell'andamento delle due loss functions e calcolarne le metriche ad essi associati.
Le metriche prese in considerazione per la valutazione delle performance sono le medesime utilizzate per misurare le performance del già citato MS-COCO, ovvero *Intersection over Union (IoU)*, *Precision* e *Recall*, utilizzate per calcolare i valori di *Average Precision* e *Average Recall*.
I dataset considerato per valutare le performance di ciascun modello sono il validation set ed il test set: pertanto sarà necessario confrontare le bounding box e le segmentation masks ottenute per ciascuna istanza con i loro valori *ground truth*.

5.2.1 Calcolo di mAP al variare della soglia minima di confidenza

In questo esempio è stata scelta una immagine dal validation set per valutare come l'output del modello varia a seconda della soglia minima di confidenza.
L'immagine selezionata contiene 5 istanze appartenenti alla classe Bottle e 5 istanze appartenenti alla classe *Jar* viste dall'alto.
Nell'esempio preso in considerazione sono state considerate, per semplicità, tre soglie minime di confidenza: 0.5, 0.6, 0.75.

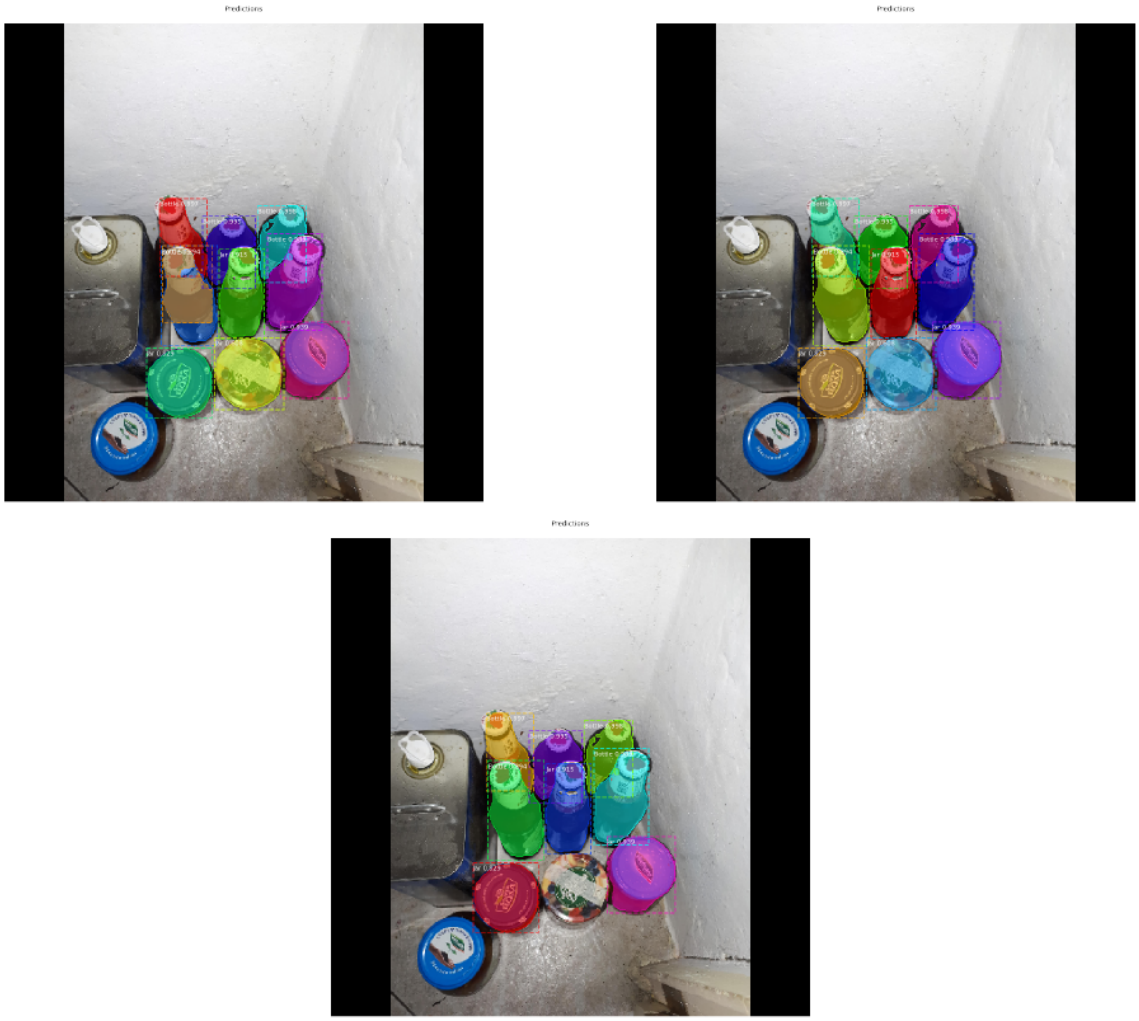


Figura 5.8: Confronto tra gli output di Mask R-CNN ottenuti al variare della soglia minima di confidenza (in alto a sinista 0.5, in alto destra 0.6, in basso 0.75)

Vediamo innanzitutto come vengono determinati i *match*, ovvero la corrispondenza biunivoca tra istanza predetta e istanza ground truth e la determinazione della curva precision-recall:

1. viene creata una matrice di dimensione $(p \times g)$ e viene determinato, per ogni coppia (i, j). Se la classe predetta è uguale alla classe *ground truth*:

   - vengono calcolati gli overlaps: è il calcolo della IoU tra due segmentation masks. In questa prima fase non è necessario stabilire se la classificazione dell'istanza sia stata eseguita correttament o meno.
     Verrà ritornata una matrice di dimensione $(p \times g)$ in cui all'elemento (i, j) corrisponderà il valore di IoU calcolato tra l'i-esima istanza predetta e la j-esima istanza *ground-truth*.
     Ad ogni istanza predetta verrà quindi associato un array contenente il calcolo della IoU per ciascuna istanza ground truth.
     Inoltre, verranno generati degli array, chiamati *pred_match* e *gt_match*, rispettivamente di dimensione p e g e le cui componenti saranno tutte inizializzate a -1.
   - Per ogni elemento predetto:
     - viene ordinato in ordine decrescente l'array contenente le IoU precedentemente calcolate;
     - vengono eliminati i valori avente un valore di IoU inferiore al valore soglia (*IoU_threshold*)
     - prendo il massimo valore di IoU (cioè il primo elemento dell'array, che è stato ordinato in precedenza): se le due classi coincidono, allora si ha il *match*, pertanto si aggiorneranno i vettori *pred_match* e *gt_match* nel seguente modo:
       * $pred\_match[i] = j$
       * $gt\_match[j] = i$

       e si esce dal ciclo. Si passa alla valutazione della successiva istanza predetta.
       Viceversa, si passa alla valutazione dell'elemento successivo dell'array: se si ha un *match* si passa alla valutazione del successivo elemento predetto. Questo procedimento andrà avanti fino a che non verrà assegnata una j-esima istanza ground truth all'i-esima istanza ground truth. Se non è possibile determinare tale corrispondenza biunivoca, le cause possono essere alcune tra le seguenti:
       * l'istanza non è stata classificata correttamente;
       * l'istanza è stata classificata correttamente ma possiede un valore di IoU minore del valore minimo
       * l'istanza non è stata individuata dalla rete

   Alla termine della procedura descritta, ogni componente del vettore *pred_match* (*ground_truth*) conterrà l'indice della componente *ground_truth* (*pred_match*) a cui è stato associato il match.
   Ad esempio, $pred\_match[0] = 3$ indicherà che una corrispondenza biunivoca

tra l'istanza predetta di indice 0 e l'istanza *ground truth* di indice 3. Se una componente del vettore *pred_match* avrà componente uguale a -1, ad esempio $pred\_match[1] = -1$, significa che non è stato possibile associare un match all'istanza predetta avente indice 1.

2. Di seguito viene riportato il codice Python utilizzato da Mask R-CNN per il calcolo di mAP:

```
    precisions=np.cumsum(pred_match>-1) / (np.range(len(pred_match
                                    ))+1)
recalls=np.cumsum(pred_match>-1).astype(np.float32)/ len(
                                    gt_match)

    precisions=np.concatenate([[0],precisions,[0]])
    recalls=np.concatenate([[0],recalls,[[1]])

    for i in range(len(precisions)-2,-1,-1):
      precisions[i]=np.maximum(precisions[i], precisions[i+1])

    indices = np.where(recalls[:-1]!=recalls[1:])[0]+1
    mAP = np.sum((recalls[indices]-recalls[indices-1])*
    precisions[indices])
```

Figura 5.9: Il seguente codice mostra come vengono calcolati i punti della curva precision-recall e il valore di mAP.

Col codice riportato in figura 5.9 è possibile ottenere due vettori *precisions* e *recalls* le cui componenti saranno ordinate, rispettivamente, in ordine decrescente ed in crescente.
Le componenti dei due vettori saranno utilizzate per generare la curva precision-recall, da cui verrà calcolato il valore di mAP secondo le formule (4.17) e (4.18).

Confidence$_{min}$ = 0.5 (50%)

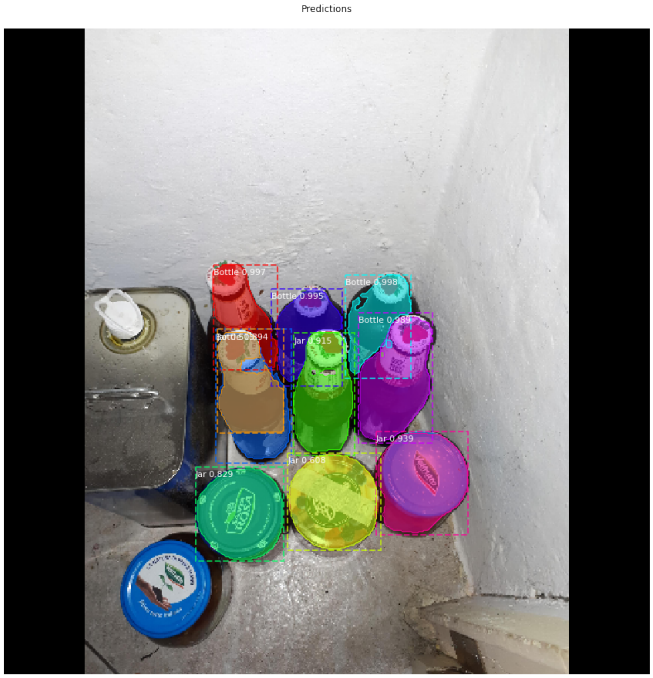


Figura 5.10: Output restituito da Mask R-CNN per un valore di soglia minima di confidenza pari a 0.5.

Come si evince dalla figura 5.10, sono state predette 10 istanze. Tra di esse, è possibile notare come un'istanza appartenente alla classe *Bottle* sia stata identificata ben due volte dal modello: la prima maschera predetta è stata classificata (correttamente) come appartenente alla classe *Bottle* con una probabilità del 89.4%, mentre la seconda è stata classificata (erroneamente) come appartenente alla classe *Jar* con una probabilità del 52%.
Un'istanza appartenente alla classe *Bottle* è stata classificata erroneamente come *Jar* con una probabilità del 92%, mentre non è stata individuata la maschera relativa ad un'istanza appartenente alla classe *Jar*.
Tutte le altre istanze predette sono risultate essere corrette.
Vediamo in maniera dettagliata come vengono calcolate le metriche necessarie per il calcolo del valore di mAP relativo all'immagine selezionata.

Figura 5.11: Matrice di confusione relativa al calcolo delle segmentation masks ottenuta con un valore minimo di confidence pari a 0.50.

L'immagine 5.11 è costituita da una matrice di dimensione $p \times g$, dove con p indichiamo il numero totale di istanze predette e con g il numero totale di istanze *ground truth*.
Oltre ai nomi delle classi associate a ciascuna istanza, sull'asse delle ordinate (*predictions*) è riportata la probabilità che l'istanza predetta appartenga effettivamente alla classe individuata. Ad esempio, un'istanza classificata come appartenente alla classe *Jar* con probabilità 0.94 (94%) indica che il modello ritiene che l'oggetto classificato sia, al 94%, una bottiglia.
All'interno di ogni elemento (p_i, g_j) è riportato il valore di IoU calcolato prendendo in considerazione l'i-esima istanza predetta e la j-esima istanza ground truth: maggiore è il valore di IoU riportato, maggiore sarà la similitudine tra la segmentation mask predetto e la segmentation mask reale.
Per calcolare i valori di precision e recall verranno considerate solamente le istanze aventi valori di IoU maggiori della soglia minima di IoU, anch'essa avente valore 0.5.
Si avranno dunque i seguenti valori di *precision* e *recall* relativi alle singole classi:

1. **Bottle**
   - *Precision* $= \frac{TP}{TP+FP} = \frac{5}{5} = 1$
   - *Recall* $= \frac{TP}{TP+FN} = \frac{5}{6} = 0.83$
2. **Jar**
   - *Precision* $= \frac{TP}{TP+FP} = \frac{3}{5} = 0.6$
   - *Recall* $= \frac{TP}{TP+FN} = \frac{3}{4} = 0.75$

Per il calcolo di mAP è necessario calcolare l'area sottesa dalla curva precision-recall. AP@50 utilizza la definizione di mAP descritta all'interno dell'equazione (4.17).
Il grafico riportato in figura 5.12 mostra l'andamento della curva precision-recall, la cui area sottesa sarà equivalente al valore di mAP relativo all'immagine selezionata. I valori intermedi di precision e recall utilizzati per il calcolo della curva sono stati ottenuti tenendo conto del numero di match individuati. Precisamente, ricordando che ogni componente i-esima dell'array *pred_match* conterrà l'indice della componente *ground truth* per cui è stato individuato un match, si ha:

$$pred\_match = [5, 3, 4, 0, 6, -1, 2, 8, 7, -1] \tag{5.1}$$

Indicando con s l'array (avente in questo caso dimensione 10) contenente la somma cumulativa data dal numero di match individuati, si avrà:

$$s = [1, 2, 3, 4, 5, 6, 7, 8, 8, 8] \tag{5.2}$$

I valori di p saranno dunque uguali a:

$$p_i = \frac{s_i}{i} \quad \forall i = 0, ...,9^1 \tag{5.3}$$

Analogamente, indicando con g_{max} il numero di istanze *ground truth* presente all'interno dell'immagine, si ha:

$$r_i = \frac{s_i}{g_{max}} \quad \forall i = 0, ...,9 \tag{5.4}$$

L'array p, conterrà i valori di precision calcolati per ciascuna istanza.Esso sarà un array non crescente in quanto $p_0 \geq p_1 \geq .. \geq p_9$.
Viceversa, l'array r, il quale conterrà i valori di recall calcolati per ciascuna istanza, sarà non decrescente ($r_0 \leq r_1 \leq ... \leq r_9$).
Inoltre, poichè precision e recall risultano definiti all'interno dell'intervallo [0,1], si avrà $p_i \in$ [0,1] $\quad \forall i = 0, .., 9$.

Una volta determinati p e r e applicati alle equazioni (4.17) e (4.18) sarà dunque possibile calcolare il valore di mAP relativo all'immagine, il quale sarà uguale a 0.767, come mostrato in figura 5.12.

[1]Si è deciso di utilizzare la medesima notazione utilizzata nel calcolo degli array in Python.

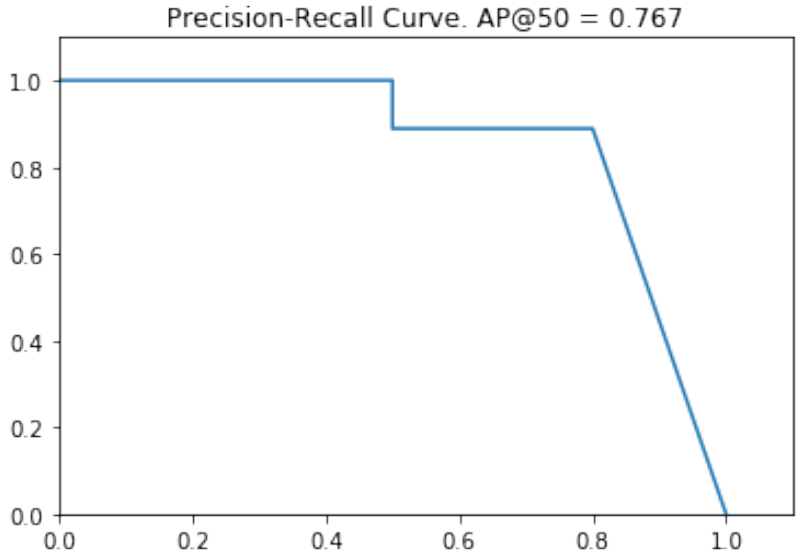


Figura 5.12: Curva precision-recall ottenuta con un valore minimo di confidence pari a 0.5.

$\textbf{Confidence}_{min} = \textbf{0.60 (60\%)}$

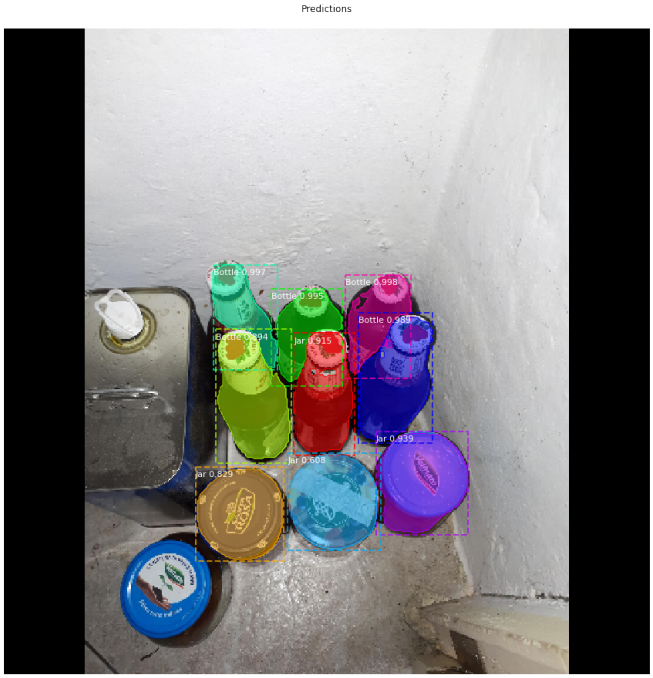


Figura 5.13: Output restituito da Mask R-CNN per un valore di soglia minima di confidenza pari a 0.6.

A differenza dell'esempio ottenuto per una soglia minima di confidenza pari a 0.5, il numero di istanze predette dal modello sarà pari a 9 (viene eliminata l'istanza appartenente alla classe *Jar* predetta con probabilità del 52%, in quanto $0.52 \leq 0.60$.).

Figura 5.14: Matrice di confusione relativa al calcolo delle segmentation masks ottenuta con un valore minimo di confidence pari a 0.60.

I valori di *precision* e *recall* relativi alle due classi saranno:

1. **Bottle**
   - $Precision = \frac{TP}{TP+FP} = \frac{5}{5} = 1$
   - $Recall = \frac{TP}{TP+FN} = \frac{5}{6} = 0.83$
2. **Jar**
   - $Precision = \frac{TP}{TP+FP} = \frac{3}{4} = 0.75$
   - $Recall = \frac{TP}{TP+FN} = \frac{3}{4} = 0.75$

Confidence$_{min}$ = 0.75 (75%)

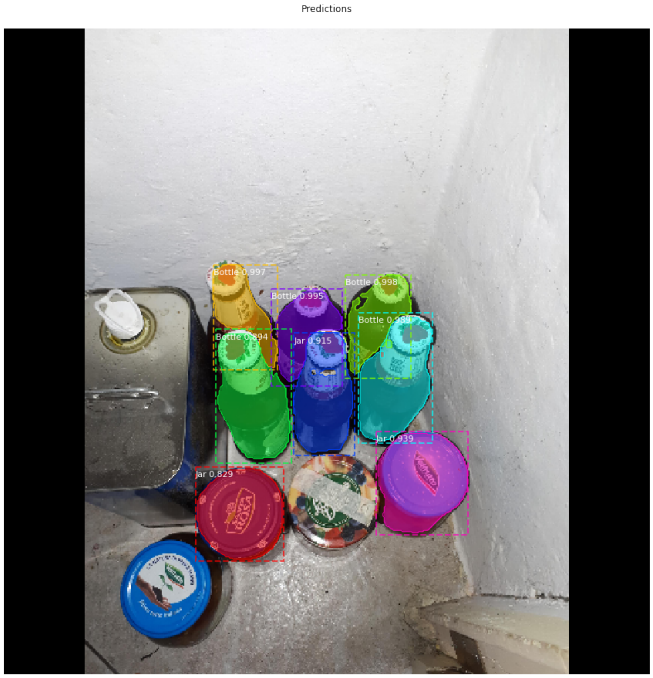


Figura 5.15: Output restituito da Mask R-CNN per un valore di soglia minima di confidenza pari a 0.75.

Dall'analisi delle predizioni riportate in figura 5.16, è possibile notare che, poichè il livello di confidenza più basso che il modello associa alle istanze individuate risulta uguale a 0.89, i risultati ottenuti saranno i medesimi variando il valore della soglia minima di confidenza all'interno del range [0.75,0.89].
I valori di precision e recall relativi alle due classi saranno uguali a:

1. **Bottle**
   - $Precision = \frac{TP}{TP+FP} = \frac{5}{5} = 1$
   - $Recall = \frac{TP}{TP+FN} = \frac{5}{6} = 0.83$
2. **Jar**
   - $Precision = \frac{TP}{TP+FP} = \frac{2}{3} = 0.66$
   - $Recall = \frac{TP}{TP+FN} = \frac{2}{4} = 0.50$

Come facilmente intuibile leggendo i valori di confidenza che la rete associa alle istanze predette nel caso precedente (figura 5.14), era lecito aspettarsi che, aumentando la soglia minima di confidenza a 0.75, il modello non avrebbe individuato l'istanza *Jar* avente valore di confidenza pari a 0.61 e, poichè a tale istanza era associato un *match*, i valori di preciaion e recall relativi alla classe *Jar* saranno influenzati dalla mancanza della predizione relativa a tale istanza.

Di seguito è riportata la matrice di confusione ottenuta dal modello utilizzando una soglia di confidenza di 0.75:

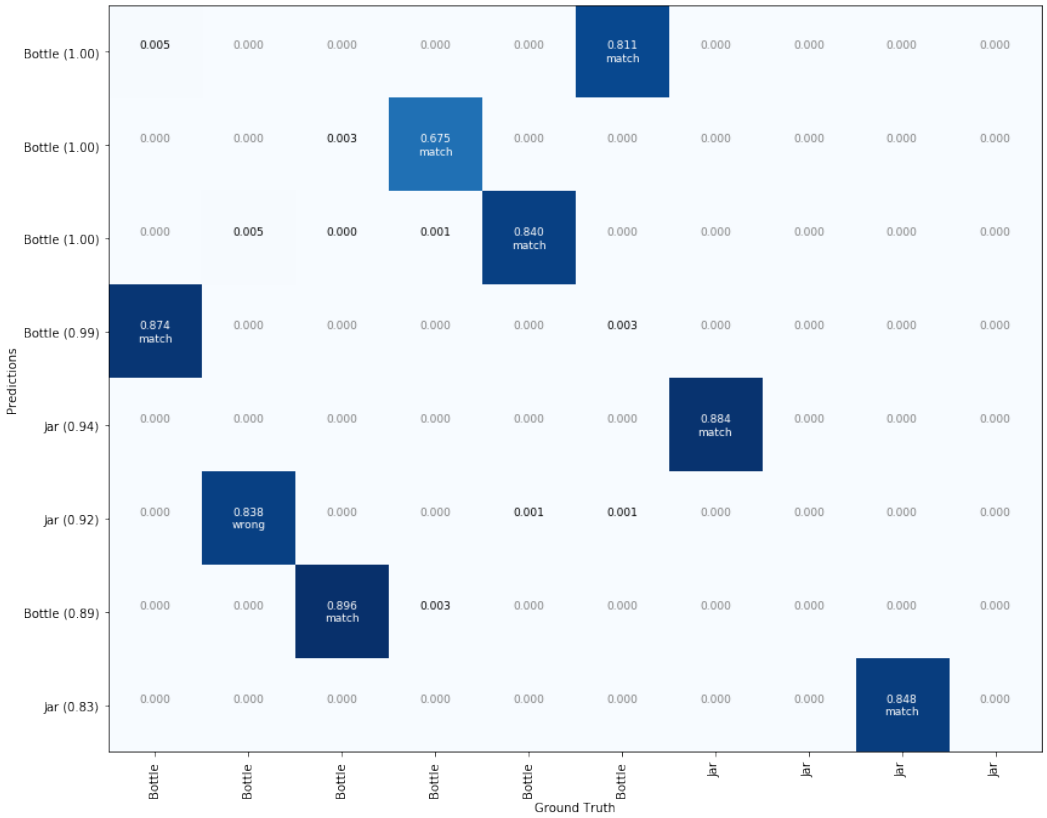


Figura 5.16: Matrice di confusione relativa al calcolo delle segmentation masks ottenuta con un valore minimo di confidence pari a 0.75.

5.2.2 Metriche di MS-COCO

Il calcolo delle metriche descritte nella sezione 4.4.4 ha permesso di ottenere i seguenti risultati:

	Model 19		Model 23		Model 25		Model 26	
	bbox	***segm***	***bbox***	***segm***	***bbox***	***segm***	***bbox***	***segm***
$AP^{IoU=[0.50:0.95]}$	0.481	0.469	**0.498**	**0.489**	0.485	0.464	0.488	0.471
$AP^{IoU=.50}$	0.639	0.643	**0.704**	**0.698**	0.666	0.677	0.664	0.670
$AP^{IoU=.75}$	0.559	0.567	**0.578**	**0.581**	0.539	0.538	0.575	0.545
AP^{small}	0.198	0.154	0.270	0.225	**0.444**	**0.402**	0.115	0.105
AP^{medium}	**0.287**	**0.301**	0.263	0.257	0.261	0.243	0.272	0.265
AP^{large}	0.503	0.489	**0.527**	**0.517**	0.516	0.497	0.525	0.505
$AR^{max=1}$	0.145	0.140	0.140	0.136	0.144	0.139	**0.152**	**0.145**
$AR^{max=10}$	0.468	0.456	0.465	0.453	0.456	0.433	**0.476**	**0.459**
$AR^{max=100}$	0.544	0.529	**0.570**	**0.554**	0.571	0.542	0.568	0.549
AR^{small}	0.243	0.200	0.371	0.329	**0.486**	**0.429**	0.229	0.214
AR^{medium}	**0.568**	**0.316**	0.285	0.283	0.279	0.265	0.283	0.278
AR^{large}	0.523	0.550	0.596	0.578	0.598	0.569	**0.601**	**0.578**

Tabella 5.6: Risultati ottenuti sul validation set.

	Model 19		Model 23		Model 25		Model 26	
	bbox	***segm***	***bbox***	***segm***	***bbox***	***segm***	***bbox***	***segm***
$AP^{IoU=[0.50:0.95]}$	0.425	0.415	**0.543**	**0.535**	0.451	0.434	0.448	0.435
$AP^{IoU=.50}$	0.577	0.573	**0.657**	**0.647**	0.639	0.646	0.617	0.625
$AP^{IoU=.75}$	0.495	0.490	0.509	**0.513**	0.496	0.495	**0.524**	0.496
AP^{small}	0.268	0.254	0.318	0.328	**0.504**	**0.499**	0.095	0.102
AP^{medium}	**0.477**	**0.446**	0.298	0.347	0.238	0.217	0.356	0.390
AP^{large}	0.440	0.428	0.468	0.459	**0.481**	**0.465**	0.476	0.460
$AR^{max=1}$	0.110	0.105	0.114	0.110	**0.120**	**0.114**	0.119	0.113
$AR^{max=10}$	0.416	0.407	0.420	0.408	0.424	0.403	**0.427**	**0.411**
$AR^{max=100}$	0.501	0.488	0.530	0.514	**0.543**	**0.516**	0.528	0.511
AR^{small}	0.310	0.310	0.355	0.327	**0.518**	**0.487**	0.125	0.122
AR^{medium}	**0.522**	**0.564**	0.314	0.332	0.270	0.305	0.422	0.415
AR^{large}	0.518	0.502	0.554	0.535	0.541	0.524	**0.556**	**0.543**

Tabella 5.7: Risultati ottenuti sul test set.

Poichè la metrica di riferimento utilizzata da COCO per la valutazione di un modello è $AP^{IoU=[0.50:0.95]}$, è possibile concludere che il modello ottimale risulta essere il modello 23, il quale ha ottenuto le migliori performance su entrambi i dataset. Queste performance sono risultate essere le migliori sia considerando i valori di IoU relativi alle bounding box, sia i valori di IoU associati alle segmentation mask.
Ulteriori analisi evidenziano come il modello 25 abbia ottenuto performance leggermente inferiori al modello 23, tuttavia esso risulta ottenere le performance migliori in

termini di AP calcolata prendendo in considerazioni solamente istanze 'piccole'.
In merito ai valori di Average Recall ottenuti, considerando il validation set, è possibile osservare come il modello 23 ne risulti avere i valori migliori in termini di AR calcolata su un numero massimo di istanze pari a 100 ($AR^{max=100}$) Le analisi relative al test set mostrano come il modello 25 abbia ottenuto i valori di Average Recall più elevati.

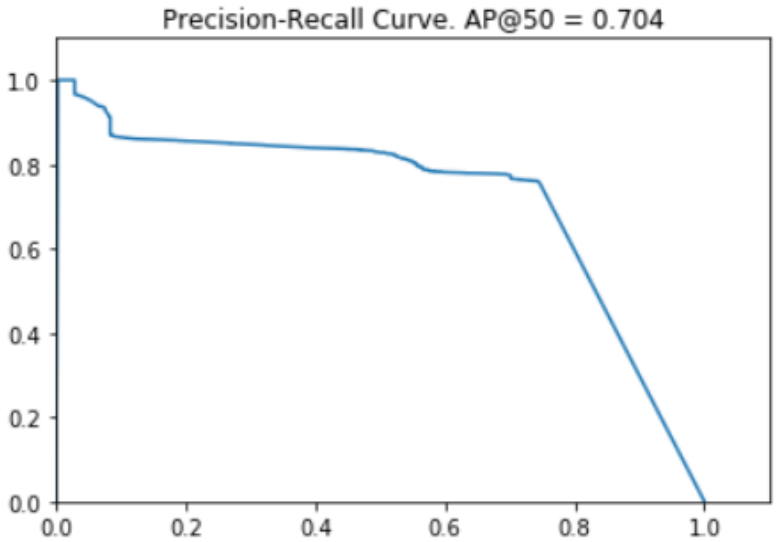


Figura 5.17: Curva Precision-Recall relativa al miglior modello ottenuto (modello 23) calcolata sul validation set.

Sulla base dei risultati ottenuti, è possibile quindi concludere dicendo che, per il tipo di problema preso in considerazione, la configurazione del modello 23 sia la preferita in termini di performance generali.
Qualora si preferisse puntare sulla capacità della rete di individuare, in particolare, istanze aventi dimensioni molto ridotte e/o che riesca ad ottenere discrete performance relativamente all'average recall, è preferibile l'utilizzo del modello 25.
Qualora sia necessario un modello in grado di identificare correttamente il maggior numero possibile di oggetti avente un'area compresa tra 32^2 e 64^2, la scelta migliore risulterebbe essere il modello 19.
In merito al modello 26, esso presenterà i valori più elevati relativi all'Average Recall e, poichè esso ha ottenuto i valori più alti in termini di $AR^{max=10}$ e AR^{large}, è possibile concludere dicendo che esso risulta preferibile quando si punta a recuperare il massimo numero di oggetti aventi grandi dimensioni e/o immagini/video in cui sono presenti al più 10 istanze.

5.3 Post-Processing

In fase di post-processing sono state valutate le performance ottenute dagli algoritmi classici di instance segmentation per determinare se, partendo dall'output delle maschere ottenute da Mask R-CNN, sia possibile o meno migliorare in maniera significativa le segmentation mask relative alle singole istanze.
Sono stati valutati i risultati ottenuti dagli algoritmi Watershed e GrabCut, le cui implementazioni sono entrambe presenti all'interno della libreria openCV.

5.3.1 Risultati ottenuti dall'algoritmo Watershed

Algoritmo Watershed

L'algoritmo Watershed si basa sul concetto che una immagine in scala di grigi può essere vista come una superficie topografica [44] in cui l'intensità dei pixel rappresenterà una zona: una regione di pixel avente un'alta intensità verrà interpretata come un rilievo (zona montuosa/collinare), mentre basse intensità verranno viste come valli, le quali verranno interpretate come regioni isolate da cui iniziare il riempimento (da cui, appunto, il nome dell'algoritmo).
Si procede a riempire ogni regione isolata con acqua di colore diverso fino a quando le regioni inizieranno a fondersi tra di loro. Una volta riempita d'acqua tutta la regione, si otterranno due gruppi differenti: i bacini idrografici e le linee spartiacque [45]. Quest'ultimo gruppo costituirà il confine tra una regione e l'altra, le quali verranno etichettate in maniera differente.

Poichè l'algoritmo Watershed proposto da openCV [44] utilizza dei marker per l'individuazione delle regioni appartenenti ad una determinata classe, un problema è la determinazione dell'area di sicuro background (i pixel non appartenenti sicuramente alla maschera dell'istanza), dell'area di sicuro foreground (i pixel che si suppongono essere appartenenti alla maschera dell'istanza) e dell'area da classificare (*unknown area*). La scelta di un'opportuna area di background/foreground influenza notevolmente la scelta dell'algoritmo in termini di classificazione dell'*unknown area*.
Per valutare correttamente i pixel dell'*unknown area* appartenenti alla maschera dell'istanza, sono state provate diverse tecniche di determinazione delle aree di background/foreground:

- *Trivial*: la maschera restituita da Mask R-CNN non viene modificata in fase di post-processing. Il valore della IoU ottenuto risulta essere uguale a quello predetto dalla rete neurale

- *Watershed 1*: (*Watershed with expansion constant* = n): la bounding box di partenza viene espansa di n pixel (ove possibile). Di tale regione, i pixel non appartenenti alla maschera faranno parte dell'*unknown area* mentre i pixel dell'immagine non appartenenti alla bounding box costituiranno la background area.

La configurazione *Watershed with expansion constant* = n tuttavia presenta un'area di background nulla nei lati in cui la differenza tra il bordo dell'immagine ed il bordo

della maschera sia minore dell'expansion constant. Tale problema è stato risolto nelle successive configurazioni:

- *Watershed 2*: (*Watershed with expansion constant = n and sure background*): risolve il problema della mancanza di area di background descritto in precedenza. Inoltre, viene aggiunta un'area quadrata di background agli estremi dell'immagine qualora la differenza tra il bordo dell'immagine ed il bordo della maschera sia minore dell'expansion constant (ad esempio, se $x_{min} - n < 0$, si porrà, per quello specifico bordo, $n = x_{min}$). Tale configurazione si limita il più delle volte a correggere i problemi di *Watershed with expansion constant = n* ma, qualora in un'immagine siano presenti solamente maschere "distanti" dai bordi dell'immagine, i risultati sono pressochè identici.

- *Watershed 3*: (*Watershed with Gamma correction*: è un'estensione di *Watershed with expansion constant = n and sure background*) con l'aggiunta di un fattore di correzione γ da applicare all'immagine originale.

- *Watershed 4:* (*Watershed with contrast and brightness*): anch'essa estende *Watershed with expansion constant = n and sure background.* La luminosità ed il contrasto dell'immagine originale vengono modificate rispettivamente di un fattore α e di un fattore β.

- *Watershed 5*: (*Watershed with more and less borders*): in questa configurazione viene modificata l'individuazione dell'*unknown area*: viene selezioata come *unknown area* gli n pixel immediatamente precedenti ed immediatamente successivi a ciascun punto del bordo.

5.3.2 Esempio di applicazione dell'algoritmo Watershed

Vediamo un esempio di applicazione dell'algoritmo Watershed in fase di post-processing. Per semplicità, è stato riportato il caso di un'immagine avente solamente un'istanza appartenente alla classe *Bottle*. Verranno confrontati i risultati ottenuti prendendo in considerazione la segmentation mask restituita da Mask R-CNN e la segmentation mask prodotta mediante la tecnica *Watershed with expansion constant* = 5 *and sure background*.

Figura 5.18: Risultato della predizione di Mask R-CNN su un'immagine contenente una singola istanza. La rete ha individuato un oggetto di classe *Bottle*.

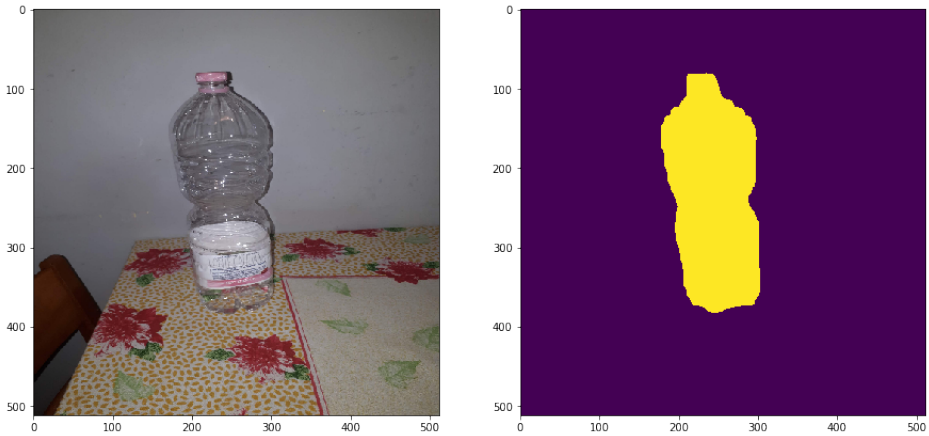


Figura 5.19: La figura mostra, a sinistra, l'immagine originale e, a destra, la maschera binaria associata all'istanza predetta da Mask R-CNN (*Trivial mask*).

Una volta ottenuta la maschera *trivial*, è stato applicato l'algoritmo Watershed: per prima cosa, è necessario determinare le regioni di sicuro *background* (non appartenenti alla maschera dell'istanza) e *foreground* (appartenenti alla maschera). L'algoritmo provvederà a determinare quali pixel della regione *unknown* apparterranno o meno alla maschera.

La maschera ottenuta applicando l'algoritmo Watershed è riportata all'interno della

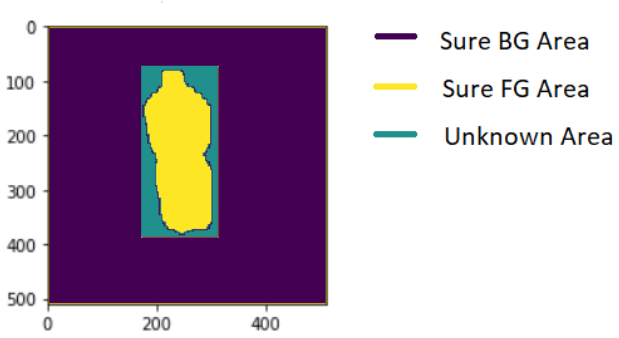


Figura 5.20: La figura mostra come viene suddivisa la maschera *trivial* al fine di poter applicare l'algoritmo Watershed.

figura 5.21, la quale mostra il confronto tra maschera *ground truth*, maschera *trivial* e maschera rifinita da Watershed.

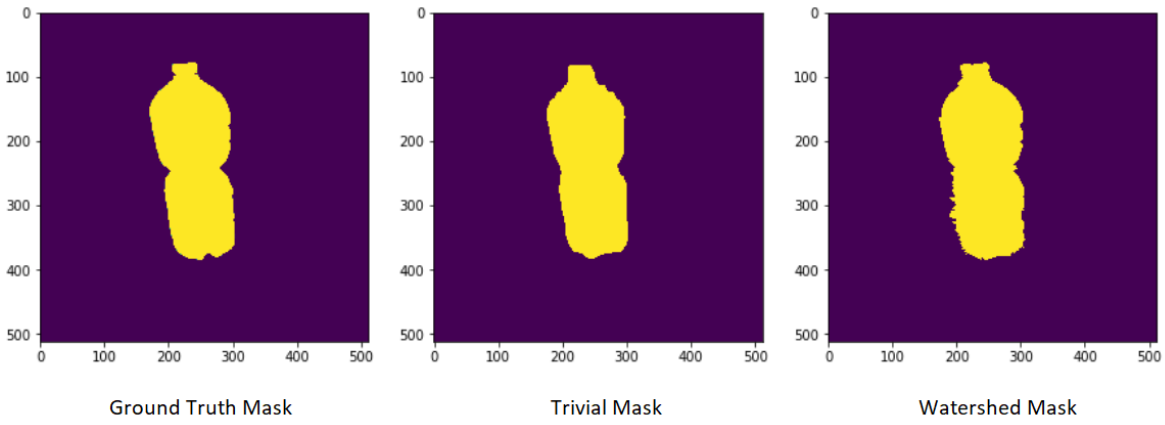


Figura 5.21: Confronto tra maschera *ground truth*, maschera *trivial* e maschera rifinita mediante Watershed.

Calcolando i valori di IoU relativi alla maschera trivial e alla maschera ottenuta mediante applicazione del Watershed, si avrà:

- $IoU_{Trivial} = 0.932$
- $IoU_{Watershed\_sure\_bg} = 0.927$

Pertanto è possibile concludere che, nonostante i due valori risultino essere molto simili tra di loro, la segmentation mask restituita dalla rete risulta rappresentare meglio l'istanza individuata.

5.3.3 Risultati ottenuti

Le tecniche di rifinitura delle maschere descritte all'interno del paragrafo 5.3.1 sono state testate sulle istanze predette da Mask R-CNN sulle immagini del validation set valutando i valori di IoU relativi alla maschera predetta dalla rete ed i rispettivi valori ottenuti mediante applicazione delle tecniche di rifinitura dell'algoritmo Waterhsed.
Innanzitutto è stato necessario individuare le corrispondenze biunivoche tra le istanze *ground truth* e le istanze predette. Una volta determinati i *match*, sono state applicate le tecniche proposte per l'algoritmo Watershed e confrontati i valori di IoU ottenuti rispetto ai valori ottenuti senza nessuna tecnica di post-processing. Proprio come nel calcolo di mAP, sono state considerate solamente istanze aventi valori di IoU > 0.5.

	Trivial	W_1	W_2	W_3	W_4	W_5
IoU	0.859	0.865	0.866	0.867	0.863	0.855
σ	0.090	0.093	0.093	0.093	0.094	0.093
σ^2	0.008	0.009	0.009	0.009	0.009	0.009
z-test	-	0.792	0.889	1.007	0.493	-0.579

Tabella 5.8: La tabella riporta il confronto tra la IoU media ottenuta considerando le maschere predette da Mask R-CNN e i valori di IoU relativi alle diverse configurazioni utilizzate nell'applicazione dell'algoritmo Watershed.

Nonostante alcune configurazioni $(W_1, W_2, W_3, W_4, W_5)$ abbiano ottenuto un incremento della IoU media calcolata per ciascuna maschera, il miglioramento ottenuto non è risultato essere in nessun caso statisticamente significativo (z-test), come si evince dal contenuto della tabella 5.8. Il test d'ipotesi è stato effettuato ad un livello di significatività $\alpha = 0.05$.
In merito al refinement ottenuto mediante algoritmo GrabCut, è stato osservato che esso richiede un costo computazionale molto più elevato comparato all'algoritmo Watershed. Pertanto, poichè l'obiettivo in fase di *refinement* delle maschere risultava quello di trovare una tecnica in grado di produrre miglioramenti significativi ed avente un costo computazionale non eccessivo, è stato scelto di non valutare le performance ottenute rifinendo le maschere con tale tecnica.
Pertanto è possibile concludere che, nel caso di studio analizzato, le tecniche di instance segmentation classiche non riescono ad ottenere risultati significativamente più rilevanti rispetto a quelli ottenuti dalla rete neurale.

6 Conclusioni

Alla luce dei risultati riportati all'interno del Capitolo 5, è stato possibile addestrare la rete Mask R-CNN per il problema di *instance segmentation* proposto.
Sono state evidenziate, inoltre, come le tecniche di *instance segmentation* classiche non migliorino i risultati ottenuti dalla rete convoluzionale.
In merito alla configurazione degli iperparametri in fase di training del modello, è stato possibile osservare come le configurazione proposte siano risultate le migliori anche con un numero minore di immagini presenti all'interno del dataset.
Complessivamente è stato osservato come i modelli ottenuti presentino un'ottima capacità di classificazione e segmentazione delle istanze ottenute per immagini contenenti un numero di oggetti non elevato: c'era da aspettarselo in quanto, maggiore è il numero di istanze presenti all'interno di un'immagine, maggiore sarà la probabilità che alcuni di questi oggetti siano sovrapposti l'uno con l'altro e, di conseguenza, complicare l'individuazione dell'istanza stessa. Ovviamente, i risultati sono ampiamente migliorabili in quanto, l'aggiunta di nuovi esempi significativi all'interno del training set, permetterà di migliorare ulteriormente le performance ottenute.
Alcune analisi più complesse che non sono state approfondite riguardano l'analisi delle performance dei modelli ottenuti sulla base di caratteristiche quali la risoluzione e la luminosità delle immagini in input.
In merito ai risultati ottenuti considerando le singole classi, è stato possibile osservare come le istanze appartenenti alla classe *Packet* fossero le più difficili in termini di predizione e segmentazione: ciò può essere dovuto alla definizione semantica che viene attribuita a tale classe. Una possibile soluzione può essere rappresentata dalla suddivisione della classe in differenti sottoclassi, ciascuna delle quali raggruppa una proprietà comune (ad esempio: pacchetto trasparente).
Poichè lo stato dell'arte è in continuo aggiornamento, l'implementazione del medesimo task utilizzando una rete neurale di implementazione più recente potrebbe condurre ad un ulteriore incremento in termini di performance dei modelli ottenuti.

Bibliografia

[1] Roberto Marmo. Intelligenza Artificiale Introduzione alle Reti Neurali. In:

[2] Jason Roell. *From Fiction to Reality: A Beginner's Guide to Artificial Neural Networks*. https://towardsdatascience.com/from-fiction-to-reality-a-beginners-guide-to-artificial-neural-networks-d0411777571b. 2017.

[3] Dario Floreano e Claudio Mattiussi. "Manuale sulle Reti Neurali". In: (gen. 2002).

[4] Ian Goodfellow, Yoshua Bengio e Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[5] Elio Piccolo. *RETI NEURALI (II PARTE)*. https://areeweb.polito.it/didattica/gcia/Materiale_Didattico/Lucidi_Corso/6_Lucidi_retineur_nuovi/Lucidi_Reti_Neurali2.pdf.

[6] Lucas Araújo. *Solving XOR with a single Perceptron*. https://medium.com/@lucaspereira0612/solving-xor-with-a-single-perceptron-34539f395182. 2018.

[7] Marvin Minsky e Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[8] Matthew D. Zeiler et al. "On rectified linear units for speech processing." In: *ICASSP*. IEEE, 2013, pp. 3517–3521. URL: http://dblp.uni-trier.de/db/conf/icassp/icassp2013.html#ZeilerRMMYLNSVDH13.

[9] Vivienne Sze et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *CoRR* abs/1703.09039 (2017). arXiv: 1703.09039. URL: http://arxiv.org/abs/1703.09039.

[10] Gareth James et al. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. ISBN: 1461471370, 9781461471370.

[11] Conor McDonald. *Machine learning fundamentals (I): Cost functions and gradient descent*. https://towardsdatascience.com/machine-learning-fundamentals-via-linear-regression-41a5d11f5220. 2017.

[12] Diego Luca Candido. *Gradient Descent o Discesa del Gradiente*. https://www.spaghettiml.com/2017/09/27/gradient-descent-o-discesa-del-gradiente/. 2017.

[13] Keremy Jordan. *Setting the learning rate of your neural network*. https://www.jeremyjordan.me/nn-learning-rate/. 2018.

[14] SAGAR SHARMA. *Epoch vs Batch Size vs Iterations*. https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9. 2017.

[15] Imad Dabbura. *Gradient Descent Algorithm and Its Variants.* https://imaddabbura.github.io/post/gradient_descent_algorithms/, 2017.

[16] David E. Rumelhart, Geoffrey E. Hinton e Ronald J. Williams. "Neurocomputing: Foundations of Research". In: a cura di James A. Anderson e Edward Rosenfeld. Cambridge, MA, USA: MIT Press, 1988. Cap. Learning Representations by Back-propagating Errors, pp. 696–699. ISBN: 0-262-01097-6. URL: http://dl.acm.org/citation.cfm?id=65669.104451.

[17] Michael A. Nielsen. *Neural Networks and Deep Learning.* misc. 2018. URL: http://neuralnetworksanddeeplearning.com/.

[18] Renu Khandelwal. *Bias and Variance in Machine Learning.* https://medium.com/datadriveninvestor/bias-and-variance-in-machine-learning-51fdd38d1f86. 2018.

[19] Yann Lecun et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86 (dic. 1998), pp. 2278 –2324. DOI: 10.1109/5.726791.

[20] Seung-Hwan Lim, Steven Young e Robert Patton. "An analysis of image storage systems for scalable training of deep neural networks". In: apr. 2016.

[21] Andrea Missinato. *Reti Neurali Convoluzionali | Il Deep Learning ispirato alla corteccia visiva.* https://www.spindox.it/it/blog/reti-neurali-convoluzionali-il-deep-learning-ispirato-alla-corteccia-visiva/.

[22] *CS231n Convolutional Neural Networks for Visual Recognition.* http://cs231n.github.io/convolutional-networks/.

[23] Craig Will. *Does pooling in convolutional networks actually work?* https://principlesofdeeplearning.com/index.php/2018/08/27/is-pooling-dead-in-convolutional-networks/.

[24] *imgaug.* https://imgaug.readthedocs.io/en/latest/.

[25] Waleed Abdulla. *Splash of Color: Instance Segmentation with Mask R-CNN and TensorFlow.* https://engineering.matterport.com/splash-of-color-instance-segmentation-with-mask-r-cnn-and-tensorflow-7c761e238b46. 2018.

[26] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *Int. J. Comput. Vision* 115.3 (dic. 2015), pp. 211–252. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0816-y. URL: http://dx.doi.org/10.1007/s11263-015-0816-y.

[27] Ross B. Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *CoRR* abs/1311.2524 (2013). arXiv: 1311.2524. URL: http://arxiv.org/abs/1311.2524.

[28] J.R.R. Uijlings et al. "Selective Search for Object Recognition". In: *International Journal of Computer Vision* (2013). DOI: 10.1007/s11263-013-0620-5. URL: http://www.huppelen.nl/publications/selectiveSearchDraft.pdf.

[29] Ross B. Girshick. "Fast R-CNN". In: *CoRR* abs/1504.08083 (2015). arXiv: 1504.08083. URL: http://arxiv.org/abs/1504.08083.

[30] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[31] Lilian Weng. *Object Detection for Dummies Part 3: R-CNN Family*. https://lilianweng.github.io/lil-log/2017/12/31/object-recognition-for-dummies-part-3.html. 2017.

[32] Kaiming He e Jian Sun. "Convolutional Neural Networks at Constrained Time Cost". In: *CoRR* abs/1412.1710 (2014). arXiv: 1412.1710. URL: http://arxiv.org/abs/1412.1710.

[33] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[34] Sabrina Sala (trad. Joyce Xu. *Guida alle architetture di reti profonde*. https://www.deeplearningitalia.com/guida-alle-architetture-di-reti-profonde/. 2018.

[35] Shaoqing Ren Jian Sun Kaiming He Xiangyu Zhang. *Deep Residual Networks with 1K Layers*. https://github.com/KaimingHe/resnet-1k-layers.

[36] Kaiming He et al. "Mask R-CNN". In: *CoRR* abs/1703.06870 (2017). arXiv: 1703.06870. URL: http://arxiv.org/abs/1703.06870.

[37] Tsung-Yi Lin et al. "Feature Pyramid Networks for Object Detection". In: *CoRR* abs/1612.03144 (2016). arXiv: 1612.03144. URL: http://arxiv.org/abs/1612.03144.

[38] Jonathan Hui. *Image segmentation with Mask R-CNN*. https://medium.com/@jonathan_hui/image-segmentation-with-mask-r-cnn-ebe6d793272. 2018.

[39] Dhruv Parthasarathy. *A Brief History of CNNs in Image Segmentation: From R-CNN to Mask R-CNN*. https://blog.athelas.com/a-brief-history-of-cnns-in-image-segmentation-from-r-cnn-to-mask-r-cnn-34ea83205de4. 2017.

[40] Adrian Rosebrock. *Intersection over Union (IoU) for object detection.* https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/, 2016.

[41] Keremy Jordan. *Evaluating image segmentation models.* https://www.jeremyjordan.me/evaluating-image-segmentation-models/. 2018.

[42] Jonathan Hui. *mAP (mean Average Precision) for Object Detection.* https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173. 2018.

[43] Waleed Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. https://github.com/matterport/Mask_RCNN. 2017.

[44] *Image Segmentation with Watershed Algorithm*. https://docs.opencv.org/3.4/d3/db4/tutorial_py_watershed.html.

[45] *IMAGE SEGMENTATION AND MATHEMATICAL MORPHOLOGY*. http://www.cmm.mines-paristech.fr/~beucher/wtshed.html.

Ringraziamenti

I miei ringraziamenti vanno innanzitutto al mio relatore, il Prof. Paolo Garza, per la sua gentilezza e professionalità mostrata nel corso di questi mesi.
Vorrei ringraziare anche Paolo Platter, CTO di Agile Lab, il quale mi ha concesso l'opportunità di sviluppare il mio lavoro di tesi presso la sua azienda.
Il ringraziamento più doveroso va a Luca Ruzzola, con il quale ho avuto la possibilità di lavorare nel corso dei miei mesi di permanenza in Agile Lab: egli si è sempre dimostrato professionale e disponibile nei miei confronti e va soprattutto a lui il merito di avermi fatto apprendere le conoscenze necessarie per la realizzazione di questo lavoro di tesi.
Ringrazio inoltre la mia famiglia per essere stati il mio vero punto di riferimento costante nel corso di questi anni, in particolare mio padre, i cui sacrifici e la sua fiducia nei miei confronti sono stati indispensabili per arrivare fino a questo traguardo.