

UNIVERSITÀ DEGLI STUDI DI TORINO

FACOLTÀ DI INFORMATICA

CORSO DI LAUREA IN INFORMATICA



Mask R-CNN: Un'applicazione nel campo della moda

PROFESSORE:

Prof. Marco Aldinucci

STUDENTE:

Edoardo Chiavazza
Matr.872450

ANNO ACCADEMICO 2019/2020

Indice

1	Introduzione	3
2	Reti neurali	5
2.1	Dalla biologia all'informatica	5
2.2	Il Percettrone	6
2.2.1	I limiti del percettrone	9
2.3	Multilayer perceptron	9
2.4	Funzione d'attivazione	10
2.5	Training	13
2.5.1	Discesa del gradiente	14
2.5.2	Backpropagation	16
3	Reti Neurali Convoluzionali	18
3.1	Operazione di convoluzione	19
3.2	Layers caratteristici di una rete CNN	20
3.2.1	Convolutional Layer	21
3.2.2	Pooling Layer	23
3.2.3	Fully-Connected Layer	24
4	Mask R-CNN	25
4.1	R-CNN	25
4.2	Fast R-CNN	27
4.3	Faster R-CNN	28
4.4	Mask R-CNN	29
4.4.1	Backbone	30
4.4.2	RoI Align: in sostituzione di ROI pooling	31
4.5	Tecniche di valutazione dei risultati per Mask R-CNN	32
4.5.1	IoT:Intersection over Union	32
4.5.2	Precision-Recall	33
4.5.3	Average Precision	34

5	Mask R-CNN applicata alla moda	35
5.1	Analisi del dataset	37
5.2	Training del modello	40
5.2.1	Data augmentation	42
5.2.2	k-Fold Cross-Validation	44
5.2.3	Controllo delle prestazioni tramite mAP	45
5.3	Inferenza del modello	48
6	Conclusioni	50

Capitolo 1

Introduzione

Le reti neurali possono essere considerate la tecnologia del secolo anche se il primo prototipo risale al 1958 con il percettrone presentato da Frank Rosenblatt. Negli anni 50' la scarsa potenza computazionale delle macchine del tempo unito allo scetticismo dei ricercatori e i pochi progressi fatti convinsero i maggiori investitori a ridurre sensibilmente il budget messo a disposizione, con l'effetto di smorzare subito l'entusiasmo per questo giovane campo di studio.

Negli anni 80' con la potenza della quinta generazione di computer permisero ad un gruppo di ricercatori giapponesi di programmare una intelligenza artificiale(IA) definita *expert system*, il cui compito era di emulare l'abilità decisionale umana.

L'*expert system* utilizzava un insieme di conoscenze rappresentate da regole *if-then* per compiere un ragionamento automatizzato con lo scopo di trovare una soluzione a problemi complessi. Il successo riscosso portò grandi investimenti che si realizzarono in IA in grado di compiere e risolvere problemi sempre più vari e intricati.

Con l'espansione e l'esperienza che si stava accumulando nel campo dell'IA unita ad alcune eccezionali soluzioni matematiche, anche altre branche dell'informatica ne giovarono. Uno di questi campi era la *computer vision* che sfruttò le *convolutional neural network*, delle reti neurali utilizzate prevalentemente nella elaborazione di immagini per *l'object detection*, *la semantic segmentation* e *l'instance segmentation*. Quando si sottopone un'immagine ad una rete neurale *l'object detection* consiste nel riconoscere gli oggetti nell'immagine, la *semantic segmentation* classifica gli oggetti in base alla categoria d'appartenenza, anche se non è in grado di visualizzare le singole istanze se queste sono molto vicine fra di loro, infine *l'instance segmentation* con un'analisi più profonda permette di segmentare i singoli oggetti.

I primi capitoli della tesi presenteranno un'infarinatura sulle reti neurali e

sulle reti convoluzionali che serviranno ad introdurre il terzo capitolo che riguarda la rete utilizzata durante il tirocinio, ovvero la Mask R-CNN. Quest'ultima è stata utilizzata per un compito *d'instance segmentation* per il riconoscimento e la classificazione di capi d'abbigliamento o accessori indossati da modelli/e e persone comuni. Il tirocinio effettuato alla DeepWare S.R.L. aveva come obiettivo la comprensione e l'utilizzo sia delle reti neurali che di tutte le tecnologie usate in questo ambito come *Tensorflow*, *Google Colab* e *Keras*. Il lavoro non si è concentrato solamente sull'aspetto del training e dei risultati della Mask R-CNN ma anteriormente è stato necessario manipolare il dataset, tramite varie librerie di Python tra cui *NumPy* e *Pandas*, per renderlo omogeneo e utilizzabile per la rete neurale. Nel capitolo 5 oltre al training e i risultati ottenuti, inoltre verrà mostrato come è stato ampliato il kernel della Mask R-CNN aggiungendo un custom callback, basato su *mean Average Precision*, per l'analisi dei risultati ottenuti ad ogni epoca dell'addestramento.

Capitolo 2

Reti neurali

"A single neuron in the brain is an incredibly complex machine that even today we don't understand. A single 'neuron' in a neural network is an incredibly simple mathematical function that captures a minuscule fraction of the complexity of a biological neuron"

Andrew Ng

2.1 Dalla biologia all'informatica

Il prototipo di una rete neurale artificiale è fortemente ispirato al modello di una rete neurale biologica anche se si è costretti ad ammettere delle notevoli semplificazioni. Le reti neurali biologiche sono composte da circa 10 miliardi di neuroni. Il neurone, l'unità funzionale delle reti nervose, è composto da

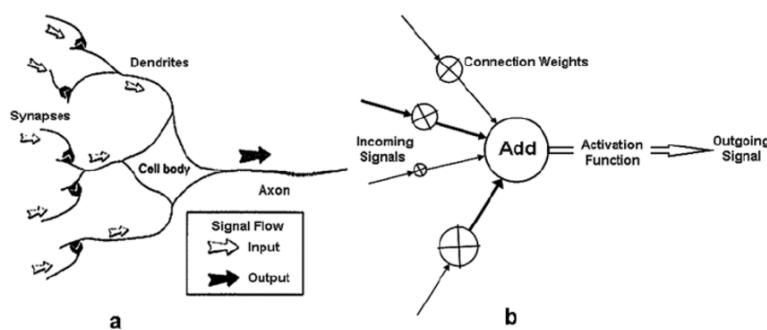


Figura 2.1: a.Neurone biologico/b.Neurone artificiale

un corpo definito *soma* e dei prolungamenti detti *dendriti* e *assoni*. I neuroni sono collegati fra loro tramite delle connessioni dette *sinaptiche* che vengono attraversate da un impulso elettrico generato dai neuroni stessi.

Se la corrente attraverso una *sinapsi* supera una certa soglia di potenziale allora il neurone ricevente trasmette sulle sue connessioni il medesimo segnale. Si possono trovare massicci parallelismi tra le reti nervose e reti neurali artificiali, siccome quest'ultime possono essere definite come modelli matematici che cercano di descrivere in modo limitato e semplificato il funzionamento della controparte biologica. In una rete artificiale si distinguono tre diversi livelli definiti come *layers*.

Il primo di questi è l'*input layer* composto dai neuroni d'input, il quale compito è ricevere informazioni esterne che possono derivare da sensori propri del sistema oppure essere costrutti software.

Il secondo è l'*hidden layer* che ha il compito di ricevere l'input dal livello precedente, computare il dato secondo il proprio compito e infine consegnarlo al livello successivo che può essere un altro *hidden layer* oppure l'*output layer*.

L'ultimo livello è il responsabile per la produzione del risultato finale e ogni rete neurale possiede uno e un solo *output layer*. Il numero di nodi è pari al numero di classi per cui si deve discriminare l'input.

Come la controparte biologica ogni neurone artificiale è connesso al livello successivo tramite dei collegamenti detti *pesi* che rappresentano la conoscenza della rete e sono definiti come $w_1 \dots w_n$.

2.2 Il Percettrone

Il percettrone, l'unità funzionale delle reti neurali artificiali, è stato descritto dal americano Frank Rosenblatt nel 1958. Secondo l'idea dello psicologo, il percettrone doveva essere una macchina più che un algoritmo. Quando l'IBM lanciò sul mercato L'IBM 704 il team di Rosenblatt implementò il neurone artificiale all'interno della nuova macchina il cui compito era il riconoscimento di immagini.

In termini moderni, l'architettura del percettrone si compone di un ingresso, un'uscita e una regola di apprendimento basata sulla minimizzazione del errore. Il neurone è un classificatore binario che implementa una funzione *threshold*, la quale mappa l'input \mathbf{x} , espresso come vettore di valori, entrante nel ingresso in un singolo valore binario $f(\mathbf{x})$ in output. Il valore uscente è descritto dalla seguente funzione:

$$F(x) = \begin{cases} 1 & \text{se } \mathbf{w} \cdot \mathbf{x} + b > 0 \\ 0 & \text{altrimenti} \end{cases}$$

- \mathbf{w} è il vettore dei pesi

- $\mathbf{w} \cdot \mathbf{x}$ è il prodotto vettoriale definito come $\sum_{i=1}^m w_i x_i$, dove m è il numero di input del percettrone
- b è il *bias*, una soglia che determina se il neurone si debba attivare. La presenza di questo parametro incrementa inoltre la flessibilità del modello, consentendogli di adattarsi meglio ai dati.

Siccome il valore restituito dalla funzione è un valore singolo, uguale ad 1 o 0, viene utilizzato per discriminare l'input.

L'algoritmo di apprendimento del percettrone converge solamente se le istanze delle due classi sono linearmente separabili, ovvero se esiste una iperpiano lineare che le separi correttamente.

L'algoritmo di apprendimento del percettrone

Essendo il percettrone un classificatore lineare, se i dati utilizzati per l'apprendimento non sono linearmente separabili il neurone non raggiungerà mai lo stato in cui è in grado di classificare il vettore di valori in input. In questo caso non si raggiunge una soluzione approssimata con una grande possibilità di errore, ma l'algoritmo di apprendimento non termina. Mentre se il set di dati è linearmente separabile, allora l'algoritmo garantisce la convergenza verso uno stato in cui è in grado con una precisione variabile di classificare il vettore in input.

L'algoritmo di apprendimento è di tipo iterativo. Per la comprensione dell'algoritmo è utile prima definire delle variabili:

Definizione delle variabili:

Le variabili utili da definire sono:

- e è il numero di *epoches of training* che effettuerà il percettrone. Il numero rappresenterà le volte che i dati verranno sottoposti alla rete per costruire i *pesi* ottimali al compito richiesto.
- lr è il *learning rate* del percettrone che può assumere valori nell'intervallo $[0.0, 1.0]$. Questo parametro definisce con quale velocità il percettrone si adatta al problema. Un valore piccolo richiederà al percettrone più epoches per raggiungere una soluzione ottima, siccome i pesi verranno aggiornati con minimi cambiamenti con il rischio però che la fase di training sia interminabile. Mentre un valore che si avvicina ad uno avrà l'effetto di modificare sostanzialmente i pesi è il risultato sarà convergere verso un soluzione sub-ottimale.

- $\mathbf{D} = \{(\mathbf{x}_j, \mathbf{d}_j), \dots, (\mathbf{x}_j, \mathbf{d}_j)\}$ è il set di dati di lunghezza j che verranno sottoposti al percepitrone, dove:
 - \mathbf{x}_j è il vettore d'input ad n-dimensioni
 - \mathbf{d}_j è il valore d'output desiderato per il vettore d'input

Mentre le *feature*¹ verranno definite come segue:

- $\mathbf{x}_{j,i}$ è il valore della *i-esima* feature del *j-esimo* vettore in input
- $\mathbf{x}_{j,0} = 1$

Per rappresentare i pesi:

- \mathbf{w}_i è il *i-esimo* peso del *vettore dei pesi*. Viene moltiplicato per il valore del il *i-esima feature* in input
- Siccome $\mathbf{x}_{j,0} = 1$, il \mathbf{w}_0 avrà il ruolo del *bias*
- $\mathbf{w}_i(t)$ è il peso *i-esimo* al tempo t , necessario per rappresentare la variazione del peso nei diversi istanti di tempo

Passi dell'algoritmo:

1. Inizializzare i pesi con un valore random piccolo oppure 0.
2. Per ogni esempio j nel training set D , vengono eseguita i seguenti passi sul input \mathbf{x}_j e sul output desiderato \mathbf{d}_j :
 - (a) Calcolare l'output attuale:

$$\begin{aligned} y_j(t) &= f[\mathbf{w}(t) \cdot \mathbf{x}_j] \\ &= f[\mathbf{w}_0(t)x_{j,0} + \mathbf{w}_1(t)x_{j,1} + \dots + \mathbf{w}_n(t)x_{j,n}] \end{aligned} \tag{2.1}$$

- (b) Aggiornare i pesi:

$$w_i(t+1) = \mathbf{w}_i(t) + \mathbf{r} \cdot (\mathbf{d}_j - y_j(t))\mathbf{x}_{j,i} \text{ per ogni features } 0 \leq i \leq n$$

L'algoritmo aggiorna i *pesi* dopo gli step 2a e 2b, applicandoli alla successiva coppia del set di dati, senza attendere che tutte le coppie nel training set siano state sottoposte agli step precedenti.

¹è una proprietà misurabile ed individuabile da una rete neurale

<i>input x</i>	<i>label</i>	<i>weights w</i>	<i>weighted sum</i>	<i>predict</i>	<i>weights w'</i>
(5, -4, 3)	+1	(0, 0, 0)	0	-1	(5, -4, 3)
(2, 3, -2)	-1	(5, -4, 3)	-8	-1	(5, -4, 3) <i>false negative (label = +1, predict -1)</i>
(4, 3, 2)	+1	(5, -4, 3)	14	+1	(5, -4, 3) <i>false positive (label = -1, predict +1)</i>
(-6, -5, 7)	-1	(5, -4, 3)	11	+1	(11, 1, -4)
					$(5 \times -6) + (-4 \times -5) + (3 \times 7)$

Figura 2.2: Esempio dell’algoritmo d’apprendimento del percettrone

2.2.1 I limiti del percettrone

I primi limiti del percettrone sono stati dimostrati nel 1969 nel libro *Perceptron* di Marvin Minsky e Seymour Papert. I due scienziati dimostrano come sia impossibile per un percettrone imparare la funzione XOR. È possibile no-

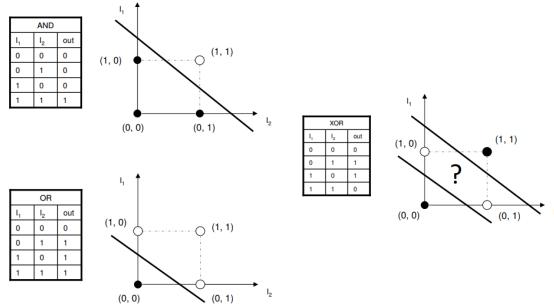


Figura 2.3: Separazione lineare delle funzioni AND,OR e XOR

tare dalla figura 2.4 che non esiste una funzione lineare in grado di separare le istanze bianche da quelle nere.

La risoluzione del problema di dati non linearmente separabili, come lo XOR, nasce dall’intuizione di unire più percetroni formando reti neurali multistrato.

2.3 Multilayer perceptron

Le reti neurali multistrato nascono dall’esigenza di poter dividere il piano delle soluzioni con più di un solo iperpiano. Come è stato descritto nel paragrafo 2.1, le reti neurali multistrato sono formate da diversi livelli composti da ognuno da uno o più percetroni. Un modello di rete neurale di questo

tipo è anche detto percepitrone multistrato (*multi layer perceptron*), o anche rete neurale feedforward, siccome l'informazione viene trasmessa dai nodi appartenenti all'input layer fino ai nodi dell'output layer senza scambio di informazioni tra nodi dello stesso strato.

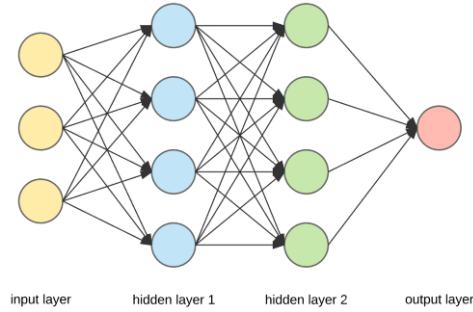


Figura 2.4: Rete neurale

2.4 Funzione d'attivazione

Nei neuroni biologici il potenziale d'azione viene trasmesso integralmente una volta che la differenza di potenziale alle membrane supera una certa soglia. Una rete neurale applica lo stesso principio solo che la risposta è adattata a seconda delle necessità, ed è determinata dalla funzione d'attivazione. La funzione descrive la correlazione tra l'input e l'output in modo non lineare, dando alla rete una maggiore flessibilità per rappresentare le relazioni tra i dati in ingresso. Se la rete fosse priva della funzione d'attivazione sarebbe assimilabile ad un modello di regressione, ovvero cercherebbe di approssimare i dati con una semplice retta ed ogni layer avrebbe lo stesso comportamento di quello precedente, quindi rendendo inutile la composizione di più perceptron. Le funzioni d'attivazione utilizzate sono molteplici, quindi verranno descritte solamente le più importanti.

Funzione a gradino unitaria

La funzione a gradino unitaria è una funzione discontinua che per valori negativi restituisce zero, mentre per argomenti positivi l'output equivale ad uno. Questo tipo di funzione non è realmente utilizzabile per la scarsa flessibilità e non è derivabile nel punto in cui cambia valore, il che è un problema siccome la derivata determina la direzione verso cui la rete neurale si orienta per aggiustare i valori. Il cambiamento brusco del valore rende difficile anche

controllare e predire il comportamento della rete, siccome una piccola modifica su un peso anche se porta miglioramento per un determinato input, può peggiorarlo per molti altri.

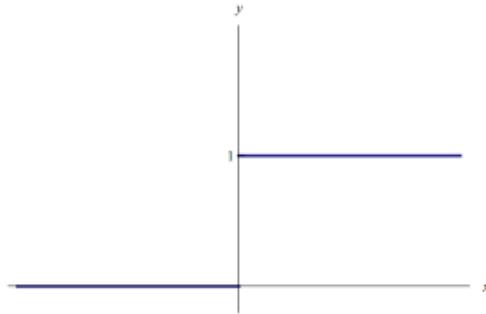


Figura 2.5: Funzione a gradino unitaria

Funzione sigmoide

Per risolvere i limiti della funzione precedente si può utilizzare la funzione sigmoide. Quest'ultima possiede delle somiglianze con la funzione a gradino, ma l'intervallo in cui cambia valore è differenziabile. Viene utilizzata spesso

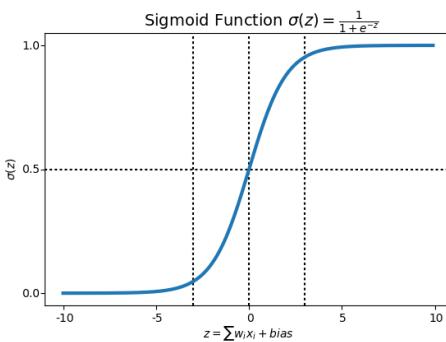


Figura 2.6: Funzione gradiente con $threshold = 0,5$

per i modelli che hanno il compito di predire una certa probabilità siccome il codominio è limitato nell'intervallo $[0,1]$. Inoltre è una funzione molto stabile anche per grosse variazioni di valori, il che la rende meno suscettibile a cambiamenti dovuti a falsi positivi² o negativi³. La funzione sigmoide soffre

²quando il modello predice per un'istanza una classe in modo erroneo

³quando il modello non predice un'istanza che ha le features di una determinata classe

del *problema della scomparsa del gradiente* e dell'*esplosione del gradiente* che possono portare ad un blocco totale della rete neurale durante la fase d'addestramento. Quindi non è una funzione che viene più utilizzata per i *layers* intermedi, ma è ancora un ottima soluzione per il livello d'output in problemi di *classificazione*

Funzione ReLU

La funzione ReLU riduce drasticamente il *problema della scomparsa del gradiente* e viene utilizzata spesso come funzione d'attivazione dei *layers* intermedi. L'output è zero per valori d'input minori e uguali a zero, mentre per valori maggiori di zero viene restituito il valore stesso. Questa semplicità la rende una funzione particolarmente appetibile nei layer intermedi, dove la quantità di passaggi e di calcoli è importante. Calcolare la derivata infatti è molto semplice: per tutti i valori negativi è uguale a 0, mentre per quelli positivi è uguale a 1. Nel punto angoloso nell'origine, invece la derivata è indefinita ma viene comunque impostata a 0 per convenzione.

Ricerche hanno dimostrato che questa funzione è la più veloce nell'addestrare con buoni risultati reti neurali vaste. Sfortunatamente anche ReLU ha delle limitazioni, infatti durante l'addestramento i pesi possono essere aggiornati in una maniera che porti i neuroni ReLU ad non attivarsi più. Questi neuroni vengono definiti "morti" e difficilmente possono ritornare attivi. Essendo disattivati il loro output sarà sempre 0, andando ad influire negativamente sull'uscita totale e sull'aggiornamento dei pesi dei neuroni vicini e di se stesso.

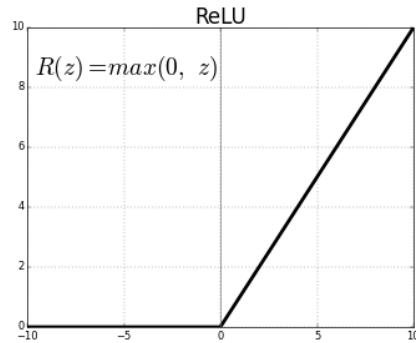


Figura 2.7: Funzione d'attivazione ReLU

2.5 Training

L’addestramento di una rete neurale permette alla stessa di trovare la configurazione dei *pesi*, in grado di massimizzare la precisione nel compito assegnatole. Per questo processo esistono tre grandi paradigmi:

1. ***Supervised Learning***: Ad ogni esempio in input è associato il risultato atteso. La rete neurale calcola il suo output, lo confronta con l’etichetta associata calcolandone l’errore e quest’ultimo verrà propagato per tutta la rete causando l’aggiornamento dei pesi. Quindi l’obbiettivo di un sistema basato sull’apprendimento supervisionato è quello di modellare una funzione in grado di “apprendere” dai risultati forniti durante la fase di addestramento, e di produrre dei risultati simili a quelli desiderati per tutte le istanze non etichettate con la soluzione. Questa tecnica viene spesso utilizzata nel riconoscimento d’immagini, ad esempio l’identificazione della scrittura manuale che si perfeziona “apprendendo” dagli esempi sottoposti dall’utente.
2. ***Unsupervised learning***: A differenza del paradigma precedente gli esempi in input non hanno la soluzione associata, quindi la rete neurale li organizzerà in classi, non note a priori, in base alla caratteristiche comuni. L’obbiettivo è “imparare” ad effettuare ragionamenti e previsioni sugli input successivi.
L’Unsupervised learning permette di eseguire compiti più complessi rispetto al paradigma precedente, siccome è in grado di riconoscere quasi tutte le caratteristiche ed organizzarle in classi senza che entrambe siano specificate dall’utilizzatore. Questo paradigma comunque ha dei limiti, il primo è la precisione che molte volte è minore rispetto ad un modello che è stato addestrato con *Supervised Learning* considerando, che è la rete a costruirsi le classi siccome non sono date in input. Il secondo limite che deriva dalla classi non conosciute a priori costringe l’utente a dover spendere tempo per interpretare e rinominare le classi create. Questo paradigma viene utilizzato ad esempio dai motori di ricerca che in base alle ricerche precedenti e tramite delle parole chiave dategli in input, restituiscono pagine con link in ordine di affinità decrescente.
3. ***Reinforcement learning***: Quest’ultimo paradigma si differenzia dai primi due perché non si hanno esempi da cui la rete neurale costruisce la propria conoscenza. Il *reinforcement learning* ha come traguardo la realizzazione di agenti autonomi, in grado di compiere azioni per raggiungere determinati obbiettivi. Per raggiungere la soluzione l’agente

deve determinare la sequenza d'azioni, influenzate dall'ambiente, per raggiungere una "ricompensa", di solito espressa tramite un valore numerico.

Esistono due tipi di *reinforcement learning*:

- (a) **Positivo:** Se l'agente compie un'azione che lo avvicina alla soluzione, allora viene incoraggiato a ripeterla. In generale il *reinforcement learning* viene utilizzato maggiormente, poiché aiuta gli agenti a massimizzare le prestazioni su un determinato compito. Non solo, l'agente apporta cambiamenti più sostenibili, i quali possono diventare soluzioni persistenti. Se viene eseguito per troppo tempo, questo rinforzo positivo può portare all'esplosione delle scelte possibili per il modello, diminuendone le prestazioni
- (b) **Negativo:** Se l'agente compie un'azione che lo allontana dalla soluzione, allora viene "redarguito" tramite la sottrazione di un valore numerico, in modo che comprenda che l'azione eseguita debba essere evitata il più possibile. Questo tipo di rinforzo è utilizzato per mantenere uno standard di prestazioni minime piuttosto che raggiungere le prestazioni massime di un modello. Il rinforzo negativo può aiutare a garantire che un modello sia tenuto lontano da azioni indesiderate, ma non fa in modo che esplori le azioni migliori per raggiungere la soluzione.

2.5.1 Discesa del gradiente

La discesa del gradiente è un algoritmo d'ottimizzazione usato per trovare in modo iterativo un minimo locale della funzione. Nel campo delle reti neurali questa tecnica viene utilizzata per trovare la migliore combinazione di *pesi* e *bias* in grado di minimizzare la *loss function* della rete neurale.

La *loss function* o *funzione di costo*, rappresenta quanto numericamente il modello abbia sbagliato nel compito assegnatogli.

MSE (Mean squadre error) è la *funzione di costo* maggiormente utilizzata nei problemi di regressione:

$$f(\mathbf{w}, \mathbf{b}) = \frac{1}{N} \sum_{i=1}^n (\mathbf{y}_i - (\mathbf{w}\mathbf{x}_i - \mathbf{b}))^2 \quad (2.2)$$

MSE utilizza come parametri w e b , entrambi vettori che rappresentano rispettivamente i pesi e bias associati ad ogni neurone. I termini $\mathbf{w}\mathbf{x}_i - b$ descrivono l'uscita generata dalla rete che sottrae l'output desiderato, calcolando l'errore per ogni singola istanza del dataset. Quindi MSE calcola

l'errore medio commesso dal modello e la discesa del gradiente la utilizza come funzione da derivare. L'idea è che se si è in grado di calcolare la derivata, quindi la pendenza della funzione, si è in grado di trovare la direzione da seguire per un minimo locale.

Se la derivata è positiva, l'aumentare il valore dei *pesi* aumenterà l'errore quindi i nuovi *pesi* saranno minori.

Se la derivata è negativa, l'incremento dei pesi porterà l'errore a diminuire, perciò i nuovi *pesi* saranno maggiori di quelli precedenti.

Se la derivata è uguale a 0, la rete neurale è già in un minimo locale, il che non richiede l'aggiornamento di alcun peso.

La derivata parziale di $\frac{df}{dw}$ è X , mentre $\frac{df}{db}$ è uguale ad 1.

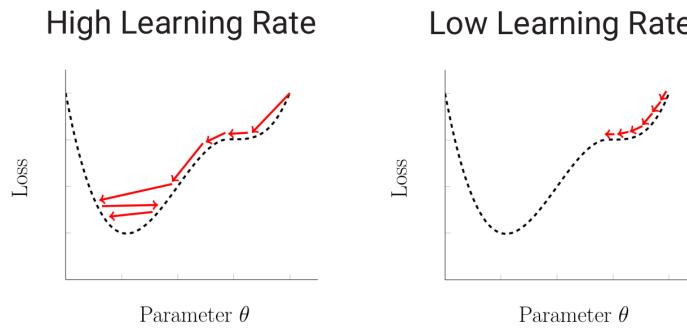
Sostituendo le derivate parziali nel *funzione di costo* si ottiene:

$$f' = \begin{bmatrix} \frac{df}{dw} \\ \frac{df}{db} \end{bmatrix} = \begin{bmatrix} \frac{1}{N} \sum -2X(y_i - (wx_i + b)) \\ \frac{1}{N} \sum -2(y_i - (wx_i + b)) \end{bmatrix} \quad (2.3)$$

I 2 che compaiono all'interno delle equazioni si possono eliminare poiché esprimono la presenza di un tasso di apprendimento due volte più grande. In definitiva, l'aggiornamento si può semplificare a:

$$\begin{aligned} m^1 &= m^0 - X(y_i - (wx_i - b)) * a \\ b^1 &= b^0 - (y_i - (wx_i - b)) * a \end{aligned} \quad (2.4)$$

La prima formula prende il nome di *delta rule*. La variabile X (il gradiente) rappresenta la direzione da seguire per minimizzare la funzione. Mentre a , il learning rate, definisce numericamente l'ampiezza del passo verso la mini-



mizzazione.

Un grande valore di questo parametro può portare ad allontanarsi dal minimo, mentre un valore molto piccolo richiederà un ampio numero di epoche d'addestramento per trovare un minimo.

Esistono due varianti della discesa del gradiente, il *Stochastic gradient descent* (SGD) e il *batch gradient descent*. Durante quest'ultimo l'errore viene esaminato di tutte le istanze d'esempio contemporaneamente, mentre per *Stochastic gradient descent* si esamina ogni errore uno alla volta.

2.5.2 Backpropagation

Nelle sezioni precedenti si è notato come i dati in input attraversino in avanti la rete neurale generando un input su cui viene calcolato un errore. La discesa del gradiente utilizzando la funzione di costo fornisce nuovi *pesi* e *bias* per aggiornare e migliorare le prestazioni della rete neurale. I nuovi parametri però non fluiscono come il vettore in entrata ma l'attraversano in direzione opposta.

Il *Backpropagation* fa fluire i nuovi pesi partendo dall'output layer con la formula:

$$\frac{\partial E}{\partial w_{i1}^m} = (\hat{y} - y) g'_o(a_1^m) o_i^{m-1} \quad (2.5)$$

Rappresenta la derivata parziale dell'errore rispetto al peso w_{i1}^m . Nella formula è composta da:

- $(\hat{y} - y)$ rappresenta l'errore
- g'_o è la derivata della funzione d'attivazione del layer di output
- a_1^m è il prodotto della sommatoria a cui viene sommato il *bias* per il nodo i nel layer m , dove m è il numero di layers che compongono la rete neurale.
- o_i^{m-1} è l'output del neurone i nel layer precedente a quello di output perciò $m - 1$

Procedendo all'indietro il layer successivo sono gli *hidden layers*, dove l'algoritmo di *Backpropagation* viene generalizzato con una formula diversa rispetto a quella vista precedentemente:

$$\frac{\partial E}{\partial w_{ij}^k} = g'(a_j^k) o^k - 1 \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1} \quad (2.6)$$

Dove:

- $1 \leq k < m$
- r^{k+1} è il numero di nodi nel layer precedente

- $1 \leq l \leq r^{k+1}$
- δ_l^{k+1} è l'errore del layer precedente

Il vantaggio di usare l'algoritmo di *Backpropagation* può essere riassunto nei seguenti punti:

- E' è veloce e semplice da programmare
- Non ha parametri a parte i numeri di input
- è un metodo flessibile in quanto non richiede una conoscenza preliminare della rete

Capitolo 3

Reti Neurali Convoluzionali

Nel campo della *computer vision*¹ l'applicazione di reti neurali *fully connected*² risulta molto inefficiente. Per ogni immagine RGB si ha bisogno di tre matrici, una per ogni colore, con righe e colonne uguali ai pixel dell'immagine scelta. Perciò un'immagine con qualità 1280×720 richiederà un'unica matrice avente dimensione $1280 \times 720 \times 3$, quindi il primo livello sarà composto da 2.764.800 nodi. Pertanto, collegare quest'ultimi con i nodi appartenenti al primo *hidden layer* saranno necessari $2.764.800 \times$ numero di nodi presenti nel primo livello nascosto. Inoltre questa tipologia di rete, considerano l'immagine diversa anche se solo si è applicata una semplice traslazione o distorsione a quella originale.

La soluzione è l'estrazione delle *features* attraverso l'impiego di *filtri* applicati a diverse zone dell'immagine. L'intuizione portò velocemente allo sviluppo

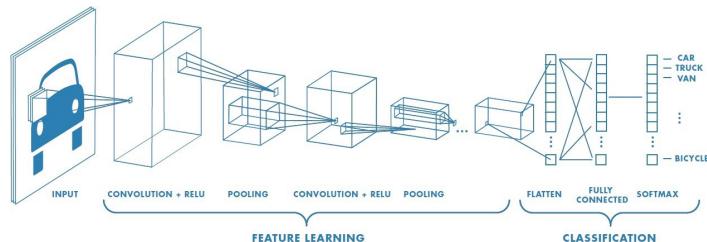


Figura 3.1: Archittetura rete convoluzionale

delle reti convoluzioni (ConvNet), conosciute anche come *shift invariant* o *space invariant artificial neural networks*(SIANN). La pre-elaborazione dell'immagine in una ConvNet è molto inferiore rispetto ad altri algoritmi di

¹è un campo interdisciplinare che si occupa del riconoscimento di immagini e video ad alto livello da parte di un elaboratore

²ogni neurone di un livello ha una connessione con ogni neurone del livello successivo

classificazione. Mentre nelle reti neurali classiche i *filtri* sono progettati a mano, con una formazione sufficiente ConvNet ha la capacità di apprenderli autonomamente. Il compito è quello di ridurre le immagini in una forma più semplice da elaborare, senza perdere caratteristiche fondamentali per ottenere una buona previsione. Assume una certa importanza non solo se si deve avere un buon modello per l'apprendimento, ma anche scalabile per enormi set di dati. Quindi una rete convuluzionale è in grado di catturare le dipendenze spazio-temporali di un'immagine attraverso dei *filtri* e la sua architettura si adatta meglio alle immagini per via del minore numeri di nodi coinvolti, e di conseguenza minori *pesi* e *bias*. Nella *computer vision* ormai sono lo strumento standard per tutti i task di *object detection*, *semantic segmentation* e *l'instance segmentation*.

3.1 Operazione di convoluzione

L'operazione di convoluzione in campo matematico ha come parametri una funzione A e una funzione B, e consiste nell'integrazione di $A \times B$, dove B viene traslato di un valore definito *stride*:

$$(A * B)(t) = \int_{-\infty}^{+\infty} A(\tau)B(\tau - t) = \int_{-\infty}^{+\infty} A(\tau - t)B(\tau) \quad (3.1)$$

L'operazione di convoluzione gode della proprietà commutativa, perciò $(A * B) = (B * A)$.

In un algoritmo ConvNet la convoluzione avviene tra l'immagine in input ed un *filtro*, entrambi sottoforma di matrici. Quindi l'equazione (3.3) assume la forma:

$$(A * B)(x, y) = \sum_{i=0}^M \sum_{j=0}^N A(x, y)B(x - i, y - j) \quad (3.2)$$

Dove A è l'immagine in input e B è il *filtro* o *kernel*. Come è possibile vedere nell'immagine 3.2, il risultato dell'operazione convoluzione è una matrice 4×4 . Il valore -7 è ottenuto dal seguente calcolo: $3 \times 1 + 1 \times 0 + 1 \times (-1) + 1 \times 1 + 0 \times 0 + 7 \times (-1) + 2 \times 1 + 3 \times 0 + 5 \times (-1)$.

L'operazione viene ripetuta spostando del valore di *stride* il *kernel*. Se la somma degli elementi del *kernel* è diversa da 0, l'immagine in input verrà schiarita o scurita.

Kernel diversi producono effetti grafici diversi, ad esempio: $\begin{bmatrix} 0 & \frac{1}{4} & 0 \\ \frac{1}{4} & 0 & \frac{1}{4} \\ 0 & \frac{1}{4} & 0 \end{bmatrix}$ sostituisce ogni punto con la media dei quattro punti posti superiormente,

inferiormente e ai lati, mentre il valore del punto centrale sarà 0. In pratica, l'immagine viene leggermente sfocata.

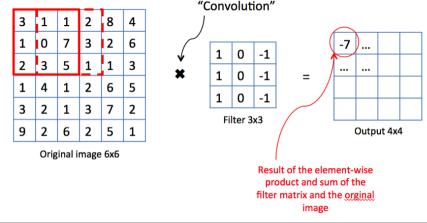


Figura 3.2: Convoluzione tra un immagine 6×6 e un *filtro* 3×3

3.2 Layers caratteristici di una rete CNN

L'architettura di una rete CNN è composta da un *layer d'input* e un *layer d'output*, tra essi si ritrovano multipli *hidden layers* con diversi compiti che verranno analizzati nel dettaglio posteriormente a questa introduzione. Una ConvNet risulta quanto meno invariante a traslazioni e distorsioni dell'immagine originale tramite tre caratteristiche:

1. *local receptive field*
2. *pesi* e *bias* condivisi
3. operazione di *pooling*

Per comprendere il livello convoluzionale bisogna introdurre il *local receptive field*.

Local receptive field

Come si è visto nell'introduzione a questo capitolo, un'immagine in input viene rappresentata come una matrice $m \times n$ dove la posizione $[m][n]$ rappresenta il valore del pixel nella stessa posizione. Perciò il livello d'input sarà composto da $m \times n$ nodi a cui ad ognuno sarà associato un pixel.

A differenza delle reti *fully-connected* ogni nodo del primo livello non sarà collegato ad ogni nodo del primo *hidden layer*, ogni neurone del primo *livello nascosto* sarà collegato solamente ad una regione dello strato d'input. Questa regione è definita *local receptive field* e per determinarla per i successivi neuroni si applica uno spostamento di valore uno dal primo *campo recettivo locale*.

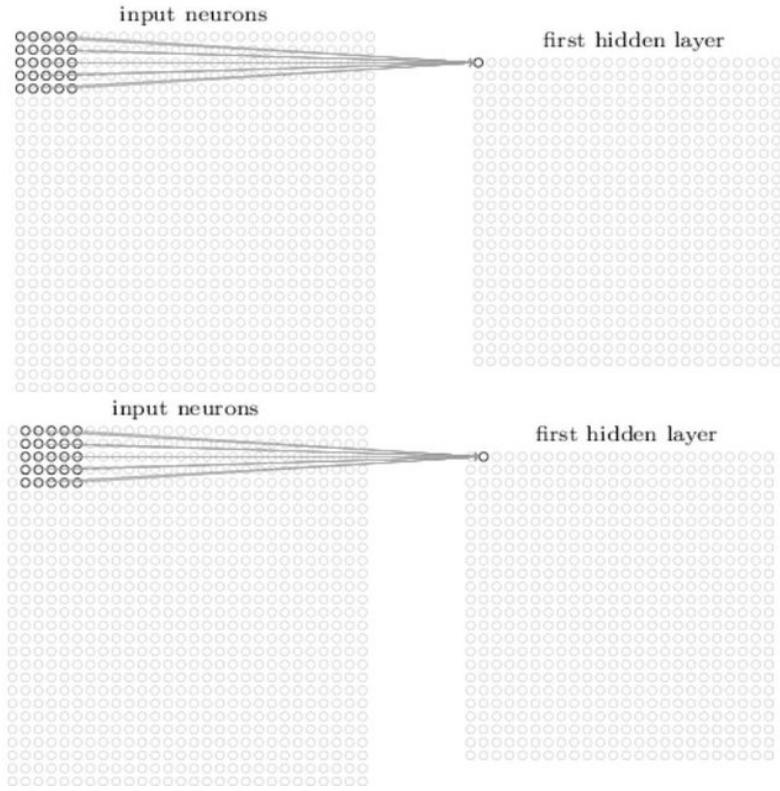


Figura 3.3: *local receptive field* di dimensione 5×5 di un primo nodo e del secondo, ottenuto mediante uno shift di valore 1

3.2.1 Convolutional Layer

Questo tipo di strato ha il compito di applicare un'operazione di convoluzione tra la matrice dell'immagine in input e diversi *filtri* della rete. L'obiettivo dell'operazione di convoluzione è estrarre le caratteristiche di alto livello come i bordi dall'immagine.

Una rete convoluzionale può possedere più *convolutional layer*, ma convenzionalmente il primo è responsabile dell'acquisizione delle caratteristiche di basso livello come bordi e i colori. Con l'aggiunta di livelli e *filtri* appositi, l'architettura si adatta anche a riconoscere le caratteristiche ad alto livello, producendo una rete che ha la comprensione completa delle *features* di un'immagine. Il valore di ogni nodo dell'*hidden layers* viene calcolato con l'equazione (3.2) e il numero di pesi utilizzati saranno solamente uguali alla dimensione del *kernel* a cui va aggiunto il *bias*.

Come è stato affermato anteriormente, le ConvNet si basano sulla condivisione dei *pesi* e del *bias*, quindi ogni nodo dell'*hidden layer* cerca di riconoscere

la stessa caratteristica in diverse aree dell'immagine. Perciò, le reti convoluzionali sono invarianti rispetto a traslazioni e distorsioni.

L'output generato dal livello convoluzionale è definito come *feature map*, la cui dimensione dipende da quattro parametri principali (definiti iperparametri):

1. *stride*
2. dimensione del *filtro*
3. numero di *filteri*
4. *padding*

L'ultimo iperparametro è uno strato da apporre all'input in modo da non perdere caratteristiche, siccome passando attraverso diversi livelli convoluzionali la dimensione dell'output diminuisce a causa della dimensione del filtro applicato. Quindi già nei primi strati della nostra rete conviene conservare il maggior numero di informazioni sull'immagine in input, in modo da poter estrarre caratteristiche di basso livello che altrimenti andrebbero perse ed impossibili da recuperare nei livelli successivi.

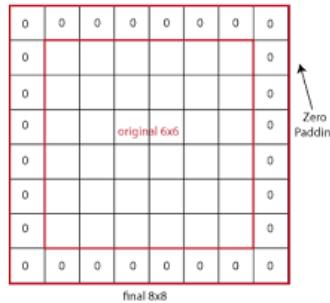


Figura 3.4: Un zero *padding* applicato ad una matrice 6×6

ReLU Layer

L'output di *convolutional layer* diventa l'input del livello ReLU, che ha scopo di introdurre la non linearità all'interno di un modello che sta compiendo, solo operazioni lineari durante i livelli convoluzionali (tramite il prodotto scalare tra il *filtro* e il campo ricettivo).

La presenza di questi livelli permette un addestramento più rapido, a causa dell'efficienza computazionale, senza impattare in modo significativo sull'accuratezza dell'inferenza del modello.

Il livello ReLU, come approfondito nella sezione delle funzioni d’attivazione, applica la funzione $f(x) = \max(0, x)$ annullando i valori negativi. Il risultato è l’aumento delle proprietà non lineari del modello senza influenzare i campi ricettivi del livello convoluzionale.

3.2.2 Pooling Layer

Lo strato di *pooling* occorre in rete neurali profonde, quindi solamente quando la mappa delle *features* risultata avere delle dimensioni tali che fluire attraverso la rete richiede molto lavoro computazionale. Lo scopo di questo livello è sostituire la matrice d’output(*features map*) del *layer ReLU* con una matrice di dimensioni ridotte.

Il ragionamento alla base di questo livello è che una volta trovata una caratteristica specifica nell’immagine di input, in cui si avrà un alto valore di attivazione, la sua posizione esatta non è importante quanto la sua posizione relativa rispetto alle altre caratteristiche.

Il *Pooling layer* riduce drasticamente la dimensione spaziale (l’altezza e la larghezza cambiano ma non la profondità, in caso di immagini RGB) della mappa delle *features* ed i requisiti computazionali per i livelli successivi.

Esistono due tecniche di pooling: *max pooling* e *average pooling*.

Max Pooling

Normalmente è la tecnica più utilizzata tra le due.

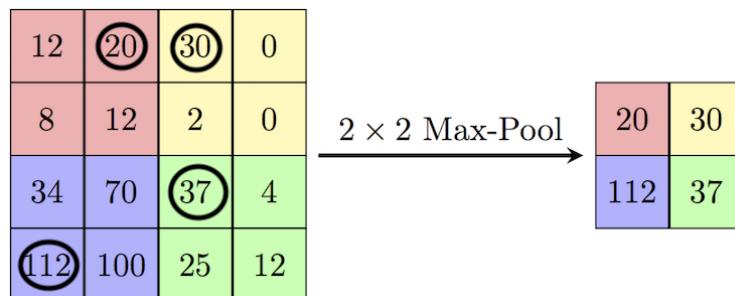


Figura 3.5: *Max pooling* applicato ad una matrice 4×4 con un *filtro* 2×2 utilizzando uno *stride* di 2

Per ogni sottoinsieme, che nell’immagine 3.5 sono le aree di diversi colori, si estrae il valore maggiore.

Average Pooling

Per ogni sottoinsieme di dimensioni del *filtro*, si fa la media dei valori e l'output sarà il valore della matrice d'output.

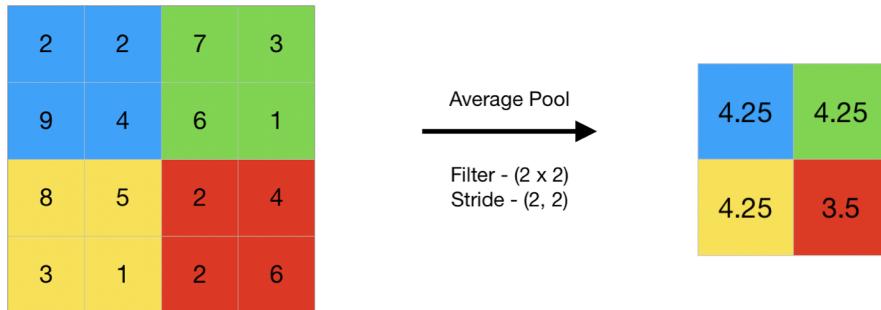


Figura 3.6: *Max pooling* applicato ad una matrice 4×4 con un *filtro* 2×2 utilizzando uno *stride* di 2

3.2.3 Fully-Connected Layer

Il *fully-connected layer* è identico al livello d'output di una qualsiasi rete neurale *fully-connected*. L'input di questo livello è una matrice di dimensione $W \times H \times D$, la quale viene trasformata tramite un'operazione di *flattening* in un vettore di grandezza N dove N è il numero di classi tra cui si deve discriminare. Ogni valore in questo vettore di dimensione N rappresenta la probabilità dell'istanza di appartenere ad una certa classe.

Fondamentalmente, un *fully-Connected Layer* riconosce quali caratteristiche di alto livello sono maggiormente correlate ad una particolare classe, calcolando i prodotti tra i *pesi* e lo strato precedente per ottenere le probabilità corrette per le diverse classi.

Capitolo 4

Mask R-CNN

Le ConvNet affrontate fino ad adesso ottengono ottime prestazioni nella classificazione di immagini d'oggetti ma per task di *instance segmentation* risultano imprecise e con un tempo di produzione dell'output non ragionevole. Il motivo principale per cui non è possibile costruire una rete convoluzionale standard seguita da uno strato *fully-connected* è che la lunghezza dello strato di output è variabile, perché il numero di istanze di interesse non è conosciuto a priori. Un primo approccio per trovare un risultato, consisterebbe in un metodo bruto di ricerca delle istanze, suddividendo le regioni in cui si trovano e sottoporre quest'ultime ad una CNN. La soluzione però è totalmente inefficiente, siccome gli oggetti di interesse potrebbero avere differenti posizioni spaziali all'interno dell'immagine e diverse proporzioni. Quindi si dovrebbe selezionare un numero enorme di regioni, facendo esplodere la complessità dal punto di vista computazionale.

Per superare il problema Ross Girshick nel 2014 propone le *Region-based Convolutional Neural Networks*(R-CNN)

4.1 R-CNN

Per ovviare al problema di individuare a priori le aree dell'immagine dove è possibile trovare le istanze viene utilizzato il *Selective search*, un algoritmo di ricerca *greedy*¹. *Selective search* estrae un numero finito, circa duemila, di regioni dall'immagine definite *region proposals*(RP). Inoltre l'algoritmo utilizza quattro valori di offset per aumentare la precisione delle RP. Ad esempio data una regione in cui l'algoritmo prevede la presenza di una persona, ma il volto all'interno della regione è tagliato a metà tramite gli offset, si ottiene

¹è un algoritmo che effettua ad ogni passo la scelta che in quel momento sembra la migliore (localmente ottima) nella speranza di ottenere una soluzione globalmente ottima

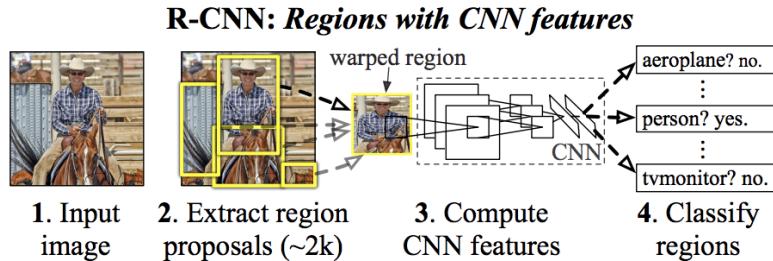


Figura 4.1: Archittettura di R-CNN

il viso nella sua interezza. L'operazione prende il nome di *padding*. Le RP vengono date in input ad una ConvNet che produrrà in output un vettore delle *features* che viene classificato tramite un algoritmo SVM.

Support Vector Machine

L'obiettivo dell'algoritmo SVM è trovare un iperpiano in uno spazio N-dimensionale (N è il numero di classi) che classifichi distintamente le istanze. Se tale iperpiano non esiste, SVM utilizza una mappatura non lineare delle istanze in una dimensione superiore. Se ne esiste più di uno, cerca quello che ha margine² maggiore tra i vettori di supporto, in modo da migliorare l'accuratezza del modello. Inoltre l'algoritmo utilizza quattro valori di offset per aumentare la precisione.

L'algoritmo Support Vector Machine, usato inizialmente per la costruzione di classificatori lineari, può essere usato per costruire modelli non lineari di dimensioni superiori. Il risultato è ottenuto tramite l'impiego di funzioni definite *kernel*, funzioni che hanno in input dati e li trasformano nella forma richiesta qualora non sia possibile determinare un iperpiano linearmente separabile, come avviene nella maggior parte dei casi.

In generale il Kernel può essere definito come:

$$K(x, y) = \langle f(x), f(y) \rangle \quad (4.1)$$

dove $\langle x, y \rangle$ indica il prodotto scalare tra vettori, mentre f è usato per mappare l'input dallo spazio n dimensionale a quello m dimensionale con $m > n$. Tuttavia, usando un kernel non lineare si ottiene un classificatore non lineare. La scelta della funzione del kernel influenza sulle prestazioni di SVM.

²è definito come la distanza tra i vettori di supporto di due classi differenti più vicini all'iperpiano. Alla metà di questa distanza viene tracciato l'iperpiano, o retta nel caso si stia lavorando a due dimensioni.

Limiti dell'architettura R-CNN

Nonostante l'architettura R-CNN elimini il problema della ricerca delle zone d'interesse, l'addestramento è costoso e lento, siccome bisogna passare attraverso questi passi che sono computazionalmente costosi:

1. Esecuzione di *selective search*, che produce circa 2000 RP per ogni immagine in input
2. Per ogni *region proposals*³ si deve generare un vettore per la classificazione, quindi N immagini * 2000.
3. Il passaggio dei vettori all'interno di una ConvNet per la classificazione.

Quindi R-CNN non è una buona candidata per il riconoscimento in tempo reale, siccome impiega 47 secondi per ogni immagine nel set di dati d'addestramento.

Oltre al peso computazionale, il *selective search* non apprende e ciò potrebbe portare alla generazione di RP pessime.

4.2 Fast R-CNN

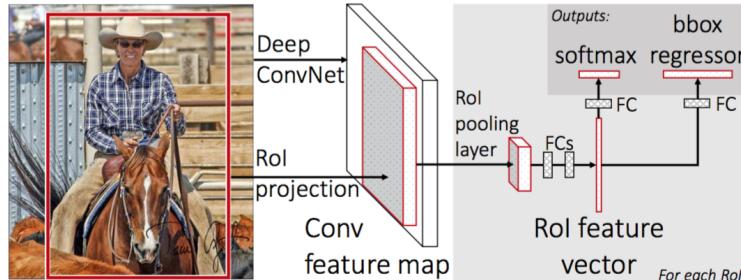


Figura 4.2: Archittettura di R-CNN

Lo stesso Ross Girshick propone la soluzione alla lentezza di R-CNN. L'approccio è molto simile ma invece di estrarre RP da ogni immagine, quest'ultima viene passata direttamente alla ConvNet che genera la mappa delle *features*. Dalla mappa vengono estratte le regioni d'interesse, poste in una *bounding box* e date in input al *RoI pooling layer* che compie un'operazione di *max pooling* e di regressione lineare per restringere la *box*. Il risultato è una matrice che data ad un *fully-connected layer* con una funzione d'attivazione

³si intende un'area avente dimensione rettangolare in cui è possibile rilevare una istanza.

softmax classifica la matrice, restituendo la classe a cui appartiene/appartengono la/le istanza/e presente/i nell'immagine in input. La ragione per cui questo algoritmo ha una complessità temporale minore della precedente è dovuto al non dover dare in input circa 2000 RP alla ConvNet, quindi i dati in ingresso risulteranno essere le sole immagini.

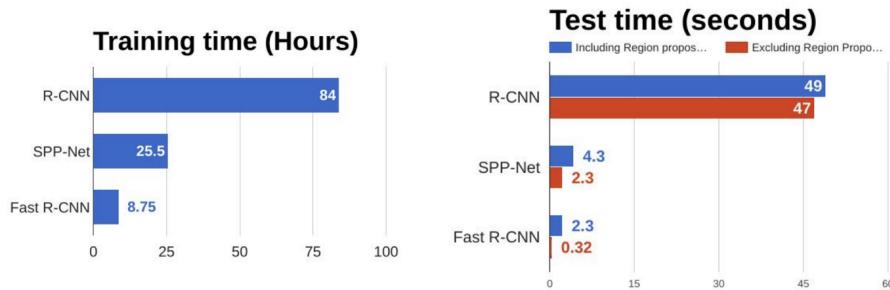


Figura 4.3: Tempo d'addestramento/test di Fast R-CNN

Come si può notare dal primo grafico, il tempo d'addestramento di Fast R-CNN viene ridotto di un fattore 10 rispetto a R-CNN. Mentre nella fase di test si può vedere come le RP rappresentino anche per Fast R-CNN il collo di bottiglia delle prestazioni

4.3 Faster R-CNN

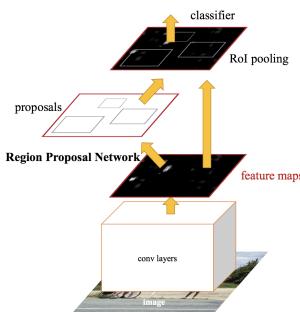


Figura 4.4: Architettura Faster R-CNN

Entrambi gli algoritmi sopracitati utilizzano *Selective search* per individuare le *region proposals*, ma l'algoritmo risulta lento riducendo le prestazioni della rete. Nel 2015 Shaoqing Ren propone un algoritmo di *object detection*

in grado far imparare alla rete come costruire le RP.

Similmente a Fast R-CNN, l'immagine viene fornita come input a una rete convoluzionale che costruisce una mappa delle *features*. Invece di utilizzare l'algoritmo di ricerca selettiva sulla mappa per identificare le RP, viene utilizzata un ulteriore rete convoluzionale, la *Region Proposal Network* (RPN), per prevedere le *region proposals*.

La RPN ha come input un'immagine e restituisce in output un insieme di RP. Nell'ultimo strato della prima CNN, presente al interno di Faster RCNN, una finestra avente dimensione $n \times n$ viene fatta scorrere lungo la *feature map* al fine di determinare dei rettangoli che contengono le *region proposal*, noti come *ancore*(anchors). Per ciascuna di queste regioni verrà associata la probabilità di contenere un'istanza, numericamente definibile tramite lo score e le coordinate dell'ancora individuata. L'operazione prende il nome di *RoI pooling*.

Una volta determinate le RP, verranno nuovamente elaborate dalla rete tramite gli strati successivi equivalenti all'archittettura di Fast R-CNN, quindi un *pooling layer* e *fully-connected layer* con una funzione d'attivazione *softmax*.

La vera differenza con Fast R-CNN ed è ciò che avvantaggia Faster R-CNN è la condivisione dei *pesi* tra la rete convoluzionale e i livelli successivi. Il risultato si può notare nel tempo d'inferenza nel grafico sottostante che mette a confronto i diversi algoritmi della classe R-CNN.

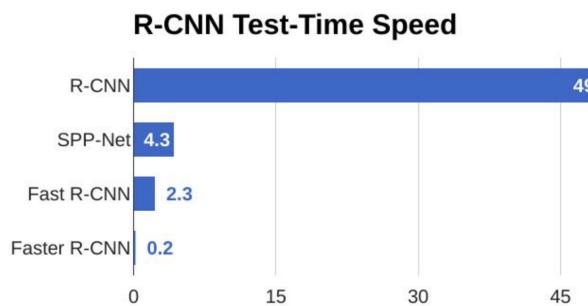


Figura 4.5: Tempo inferenza di R-CNN/ Fast R-CNN/ Faster R-CNN

4.4 Mask R-CNN

Mask R-CNN è una rete neurale utilizzata per scopi di *instance segmentation* e si differenzia da Faster R-CNN dall'aggiunta di un terzo branch, il cui compito è di generare una maschera più o meno precisa dell'istanza. Quest'ultima prende forma dalle regioni d'interesse, partendo da una dimensione

solitamente di 28 x 28, per infine raggiungere la grandezza della *bounding box* in cui l'istanza è contenuta. Perciò se l'output di Faster R-CNN è per ogni oggetto candidato il nome della classe e la *bounding boxes*, l'output di Mask R-CNN sarà per ogni oggetto candidato il nome della classe e la *bounding boxes* ma anche la maschera associata. L'algoritmo Mask-RCNN si può

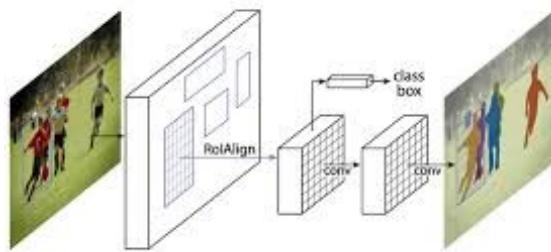


Figura 4.6: Architettura Mask-RCNN

dividere prevalentemente in due stage, siccome hanno compiti diversi. Il primo rappresenta l'individuazione delle RP e il secondo classifica le istanze, entrambi gli stage sono collegati al *BackBone*

4.4.1 Backbone

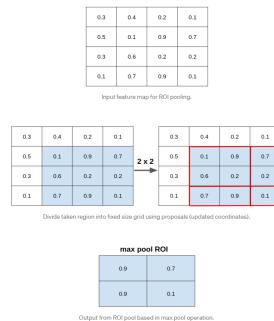
Il termine *backbone* si riferisce alla parte di rete (a sua volta è una rete) della famiglia R-CNN che si occupa dell'estrazione delle *features*. Esistono diverse architetture *backbone*, di solito si sceglie quella che si adatta meglio al modello e al compito. Le più famose sono *resnet xception* e *mobilenet*. Per Mask-RCNN, gli algoritmi *backbone* che hanno ottenuto migliori prestazioni sono *ResNet* e *ResNeXt*, entrambe aventi 50 o 101 layers. Per ottenere un'estrazione delle *features* più precisa si può unire ad un primo *backbone* una seconda specifica rete convoluzionale la *Feature Pyramid Network*(FPN).

L'architettura di FPN presenta due percorsi, il primo bottom-up e il secondo top-bottom e due connessioni laterali. Il percorso bottom-up può essere implementato con una qualsiasi ConvNet, solitamente ResNet, il cui compito è estrarre le *features* dalle immagini non elaborate. Mentre il compito del percorso dall'alto verso il basso è di generare una mappa delle *features* su diverse scale, di dimensioni simili al percorso dal basso verso l'alto. Le connessioni laterali operano trasformazioni di convoluzione e di *pooling* tra i diversi output dei due percorsi.

4.4.2 RoI Align: in sostituzione di ROI pooling

Nell'operazione di *ROI pooling*, RPN restituisce le RP e tutte le regioni rappresentano gli offset per ogni *ancora*. La rete utilizza gli offset per ricavare i riquadri che delimitano le istanze, ma le coordinate vengono restituite sulla base della dimensione dell'immagine originale. Dopodiché utilizzando le *bounding boxes* precedentemente calcolate, viene ritagliata la *region proposals* dalla mappa delle *features*. Ma la mappa delle caratteristiche nelle varie operazioni di convoluzione è stata ridotta k volte rispetto all'immagine originale, significa che ogni coordinata dei riquadri deve essere diminuita a sua volta di k volte. In primo luogo, l'operazione di *ROI pooling* divide ogni coordinata per k e prende solamente la parte intera. Dopo di che con le nuove coordinate relative alla dimensione della mappa delle caratteristiche, da quest'ultima viene ritagliata l'area in cui si è individuato un oggetto. In seguito per ottenere un output di dimensione fissa si applica alla parte ritagliata un'operazione di *max pooling*.

RoI pooling è utilizzata da Fast R-CNN e Faster R-CNN.



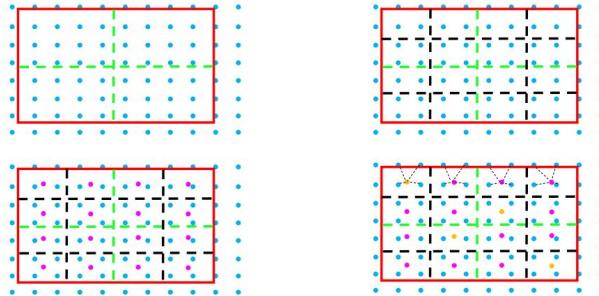
Un'importante novità introdotta da Mask R-CNN è l'utilizzo del *RoI⁴ Align* in sostituzione del *RoI Pooling* per l'estrazione delle *region of interest* di una *feature map*.

ROI align divide ogni coordinata delle *bounding boxes* per k , ma non estrae la parte intera. Perciò essendo i pixel numeri interi, le nuove coordinate non si possono usare siccome sono numeri decimali. Ciononostante la soluzione è dividere la parte ritagliata in una griglia, scegliere quattro punti per ogni area della griglia usando l'*interpolazione bilineare*. L'output dell'operazione sarà una matrice i cui valori sono il massimo o la media dei quattro punti di ogni zona.

Mask R-CNN non utilizza *ROI pooling* siccome è molto approssimativa, infatti per un compito di *instance segmentation* diviene un problema avere

⁴Region of Interest: regioni in cui è probabile trovare un'istanza da classificare

degli offset approssimativi durante la previsione della maschera.
I risultati di Mask R-CNN con ROI sono molto più precisi rispetto a quelli ottenuti con *RoI pooling*.



4.5 Tecniche di valutazione dei risultati per Mask R-CNN

Esistono diverse tecniche per valutare le performances di Mask R-CNN, queste sono le principali.

4.5.1 IoU:Intersection over Union

IoU è una metrica per la valutazione dell'accuratezza di Mask R-CNN su un set di dati. Questa tecnica calcola la sovrapposizione tra la maschera *ground-truth*⁵ e la maschera predetta dal modello. Perciò, l'output è il rapporto tra l'area d'intersezione e l'area di unione delle due maschere ed avrà valore compreso nell'intervallo [0.00, 1.00].

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

Esaminando l'equazione, si può notare che questa metrica può essere usata

⁵rappresenta un'informazione che il modello utilizza per l'addestramento

anche per tutta la famiglia delle reti R-CNN. Difatti, al numeratore si trova l'area di sovrapposizione tra il riquadro previsto dal modello e il riquadro *ground-truth*.

Il denominatore è l'area di unione, o più semplicemente, l'area racchiusa sia dal riquadro di delimitazione previsto che dal riquadro *ground-truth*.

IoT serve per premiare il modello quando le *bounding boxes* predette si sovrappongono, con risultati più o meno buoni, alle *bounding boxes ground-truth*.

Rimane comunque, estremamente improbabile che le coordinate (x, y) del riquadro calcolato corrispondano esattamente alle coordinate (x, y) del riquadro dato, a causa delle diverse operazioni per l'estrazione delle *features*.



Come si può notare, i riquadri che si sovrappongono meglio agli altri, hanno punteggi più alti rispetto a quelli che si sovrappongono meno.

Perciò *Intersection over Union* risulta essere una metrica eccellente per la valutazione di modelli che lavorano su set di dati custom.

4.5.2 Precision-Recall

Precision-Recall non calcolano la precisione delle *bounding boxes*, ma della capacità del modello di classificare in modo esatto le istanze individuate. In un compito di classificazione viene dato in input una istanza x e in output y che rappresenta la classe predetta.

Il risultato y può rientrare nelle seguenti casistiche:

- *True Positive* (TP): y è una istanza che viene classificata correttamente dal modello come appartenente ad una certa classe.
- *True Negative* (TN): y è una istanza che viene classificata correttamente dal modello come non appartenente ad una certa classe.
- *False Positive* (FP): y è una istanza che viene classificata in maniera non corretta dal modello come appartenente ad una certa classe.

- *False Negative* (FN): y è una istanza che viene classificata in maniera incorretta dal modello come non appartenente ad una certa classe.

Perciò la metrica *precision* si può calcolare come:

$$\frac{TP}{TP + FP} \quad (4.2)$$

L'equazione definisce la percentuale di positivi classificati correttamente dal modello.

La metrica *recall* viene rappresentata dall'equazione:

$$\frac{TP}{TP + FN} \quad (4.3)$$

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

4.5.3 Average Precision

Utilizzando le due misure precedentemente discusse, non possono dare la sicurezza di avere costruito un buon modello, siccome se la *precision* aumenta di valore, la metrica *recall* decresce e viceversa. Perciò si valuta l'area risultante dall'integrale:

$$\int_0^1 p(r)dr \quad (4.4)$$

L'area al di sotto della funzione (4.4) è detta *average precision*(AP) ed assumerà un valore nell'intervallo [0.0, 1.0].

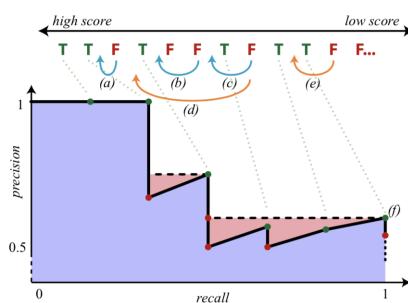


Figura 4.7: Esempio di *average precision*. Nelle ascisse i valori relativi alla recall e nelle ordinate i valori relativi alla precision.

La media dei valori ottenuti dall'*average precision* calcolati per ogni classe costituisce la *mean average precision*(mAP).

Capitolo 5

Mask R-CNN applicata alla moda

L'obbiettivo del tirocinio è stato la costruzione di un modello *custom* di Mask R-CNN. Il compito proposto è stato di *instance segmentation* su abiti e accessori indossati da modelli e persone comuni. La versione da cui si è partiti di Mask R-CNN è stata ottenuta da github tramite:

¹ https://www.github.com/matterport/Mask_RCNN.git

La repository, implementata in *Python*, è stata ampliata con una funzione di normalizzazione del dataset dato in input.

MS COCO: pesi iniziali

Siccome l'addestramento del modello partendo da *pesi* casuali avrebbe richiesto una quantità di tempo superiore al tirocinio, ed avrebbe portato difficilmente a risultati sufficienti nel breve periodo, sono stati usati dei *pesi* già addestrati, compiendo un'operazione di *transfer learning*.¹ I pesi sono stati ottenuti dalla seguente repository:

¹ https://github.com/matterport/Mask_RCNN/releases/download/v2.0/mask_rcnn_coco.h5

L'addestramento per avere i seguenti pesi, è stato eseguito sul database MS COCO. Quest'ultima è di proprietà di Microsoft ed ha come obiettivo fornire una moltitudine di dati diversi per l'addestramento supervisionato di modelli, i cui compiti sono *object detection* e *instance segmentation*.

Google Colab: configurazione GPU

Il training della rete neurale non è stato effettuato in locale per la scarsa potenza computazionale, ma si è preferito appoggiarsi alla tecnologia *cloud*

¹consiste nell'effettuare l'addestramento partendo da pesi già allenati.

di Google: *Google Colab*.

"Colab" utilizza *Jupyter notebook*², per permettere di scrivere ed eseguire codice Python nel browser senza nessuna configurazione necessaria e soprattutto con accesso gratuito alle GPU per il training. *Google Colab* sfrutta come GPU, in ordine di potenza crescente, le Nvidia K80s, T4s, P4s e P100s. La configurazione utilizzata durante il tirocinio comprendeva l'utilizzo di una forma di runtime con RAM elevata ed una *accelerazione hardware* tramite GPU, in modo da sfruttare la massima potenza computazionale messa a disposizione.

Il miglioramento dato dalla GPU si può notare eseguendo il seguente codice in "Colab":

```
1 %tensorflow_version 2.x
2 import tensorflow as tf
3 import timeit
4
5 device_name = tf.test.gpu_device_name()
6 if device_name != '/device:GPU:0':
7     print(
8         '\n\nThis error most likely means that this notebook is
9 not ,
10     configured to use a GPU. Change this in Notebook
11 Settings via the '
12     'command palette (cmd/ctrl-shift-P) or the Edit menu.\n
13 ')
14     raise SystemError('GPU device not found')
15
16 def cpu():
17     with tf.device('/cpu:0'):
18         random_image_cpu = tf.random.normal((100, 100, 100, 3))
19         net_cpu = tf.keras.layers.Conv2D(32, 7)(random_image_cpu)
20         return tf.math.reduce_sum(net_cpu)
21
22 def gpu():
23     with tf.device('/device:GPU:0'):
24         random_image_gpu = tf.random.normal((100, 100, 100, 3))
25         net_gpu = tf.keras.layers.Conv2D(32, 7)(random_image_gpu)
26         return tf.math.reduce_sum(net_gpu)
27
28 # We run each op once to warm up; see: https://stackoverflow.
29     com/a/45067900
30 cpu()
31 gpu()
32
33 # Run the op several times.
```

²Applicazione web open source che permette la creazione di documenti, i quali contengono codice, equazioni, grafici e anche testo

```

30 print('Time (s) to convolve 32x7x7x3 filter over random 100
      x100x100x3 images '
31     '(batch x height x width x channel). Sum of ten runs.')
32 print('CPU (s):')
33 cpu_time = timeit.timeit('cpu()', number=10, setup="from
      __main__ import cpu")
34 print(cpu_time)
35 print('GPU (s):')
36 gpu_time = timeit.timeit('gpu()', number=10, setup="from
      __main__ import gpu")
37 print(gpu_time)
38 print('GPU speedup over CPU: {}'.format(int(cpu_time/
      gpu_time)))

```

Nella immagine sottostante si può notare la differenza di tempo tra una CPU e GPU in un'operazione di convoluzione, utilizzando un filtro di dimensioni $32 \times 7 \times 7 \times 3$ su 100 immagini di grandezza $100 \times 100 \times 100 \times 3$.

```

Time (s) to convolve 32x7x7x3 filter over random 100x100x100x3 images (batch x height x width x channel). Sum of ten runs.
CPU (s):
3.862475891000031
GPU (s):
0.10837535100017703
GPU speedup over CPU: 35x

```

5.1 Analisi del dataset

I dati per condurre un addestramento sono stati forniti da *Kaggle*, un sito di competizioni di *machine learning*. L'obbiettivo di questa fase embrionale è fondamentalmente capire la migliore configurazione di Mask R-CNN per il compito proposto.

Il dataset è stato fornito di 40675 immagini con labels per l'addestramento e 4520 immagini per valutare il modello nel momento dell'inferenza. Le labels delle immagini di training sono descritte dal file "label_descriptions.json", il cui contenuto è il numero di classi e gli attributi tramite per cui la rete deve discriminare le istanze.

Il primo lavoro svolto è stato il cercare di comprendere la distribuzione delle classi per valutare il livello di bontà del dataset. Come si può notare nella prima immagine della figura 5.1, il dataset non è omogeneo. Perciò istanze come gli ombrelli, che sono poco presenti nelle immagini di test hanno una bassa probabilità di essere ben assimilati dalla rete.

Per fortuna o sfortuna, il set di foto su cui si eseguirà il test ha una distribuzione di classi simile al precedente. Quindi i risultati sulle immagini di test potranno considerarsi veritieri.

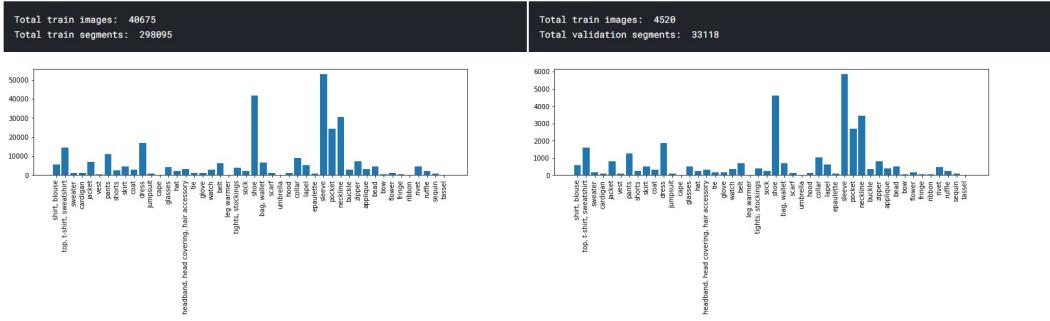
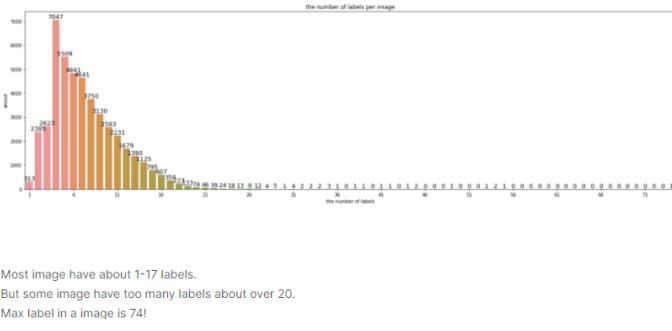


Figura 5.1: Distribuzione delle labels nel training set / Distribuzione delle labels nel test set

Il secondo compito è stato quello di conoscere quale è la distribuzione di labels per ogni immagine, per comprendere se si stesse lavorando su dei dati *multi-label* o *single label*:



Dal grafico si evince che si sta lavorando con dati *multi-label*

Da Kaggle a Google Colab

All'inizio del tirocinio si pensava di utilizzare la potenza computazionale in cloud di Kaggle, ma la piattaforma richiedeva un abbonamento mensile. Però si è optato per una soluzione free come Google Colab. L'unica limitazione sono i 15 GB di spazio di memorizzazione, mentre in Kaggle non si ha un limite. Il problema è sorto dalla dimensione del dataset(35 GB), il chè ha richiesto un lavoro di riduzione delle immagini cercando di mantenere la presenza di ogni label. Il codice implementato per questo compito sfrutta le librerie *Numpy* e *Pandas*, le quali facilitano la manipolazione di dati.

La funzione è visibile di seguito:

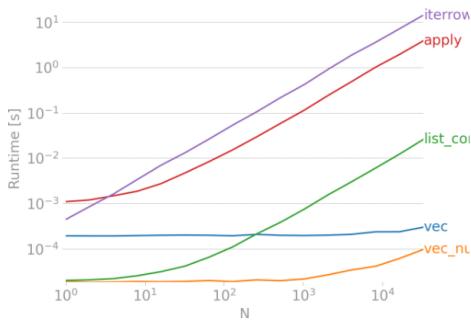
```
1 import pandas as pd
2 import json
```

```

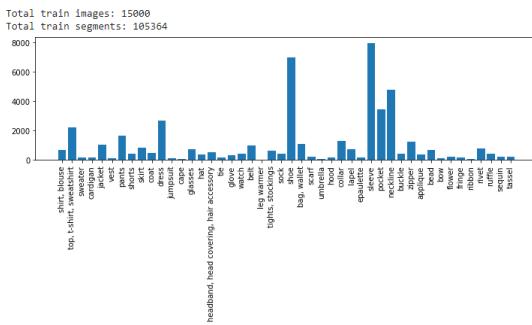
3 import shutil
4 import numpy as np
5 class ReduceDataFrame():
6     i = 0
7     df1 = pd.DataFrame()
8     label_id = [x['id'] for x in label_descriptions['attributes']
9                 ,]]
10    fields = ['ImageId', 'EncodedPixels', 'Height', 'Width', ,
11               'ClassId']
12    df = pd.read_csv(PATH_TO_CSV, skipinitialspace=True,
13                      usecols=fields)
14    df2=df['ClassId'].str.split("_",expand = True)
15    df['index'] = df.index.values
16    df2['index'] = df2.index.values
17    df2 = pd.merge(df, df2, on="index")
18    df2=df2.drop(columns=['ClassId', 'index','EncodedPixels', ,
19                   'Height', 'Width'])
20    for num_img in range(0, MAX_IMAGE):
21        for id_category in label_id:
22            arr = df2.to_numpy()
23            find_image = False
24            for n_row in range(0,len(arr)):
25                for n_column in range(1, np.size(arr, 1)):
26                    if arr[n_row][n_column] == str(id_category)
27                    :
28                        i+=1
29                        image_id = arr [n_row][0]
30                        print(arr[n_row])
31                        find_image = True
32                        df1=df1.append(df.loc[df['ImageId'] ==
33                               image_id])
34                        shutil.copyfile((PATH_TO_IMG + image_id)
35                               ,(PATH_TO_IMG_REDUCED + image_id))
36                        df2 = df2[df2['ImageId'] != image_id]
37                        break;
38                        if(find_image == True):
39                            break;
40    df1.to_csv(PATH_TO_CSV_REDUCED)

```

Il codice si compone in due parti principali. La prima è compresa tra le linee 10-15, dove le istanze *single label* vengono convertite in dati *multi-labels*. Nella seconda parte tramite due cicli si cercano le immagini che hanno una determinata label, in modo da avere un set di dati che prima di tutto contenga tutte le label se secondo che queste abbiano una distribuzione omogenea. Nella linea 18, il *Pandas* dataframe viene convertito in *Numpy* array, siccome la ricerca di un elemento all'intero della seconda struttura dati risultata più veloce della prima per dati *multi-labels*.



Come è visibile le operazioni tramite vettori numpy risultano molto più veloci che qualsiasi altro metodo per iterare su un dataframe. Il risultato della applicazione del codice precedente ha prodotto i seguenti risultati:



Sono state scelte 15000 immagini su 45195 del dataframe originale ed è come possibile vedere, è stata mantenuta la stessa distribuzione delle labels iniziale.

5.2 Training del modello

Il training è la parte più lunga e delicata del processo di creazione del modello. Mask R-CNN ha molti parametri, la cui variazione può causare miglioramenti o rallentamenti nel tempo di raggiungimento di una soluzione. L'obiettivo di questa fase è la creazione di un modello che apprenda in modo corretto le istanze, pur mantenendo la capacità di generalizzare le conoscenze, al fine di evitare l'*overfitting* sui dati d'addestramento. L'*overfitting* accade quando il modello si "abita" ai dati in ingresso ed invece di migliorare con l'aumentare delle epoche di training, la sua precisione decresce a causa di non sapere generalizzare sulle istanze di test. Un primo indizio che il modello sta andando verso questa problematica, si può notare se la *loss function* relativa alla *classification* (o *classification loss*) è all'incirca della stessa

al *validation set* (*validation loss*) cresce all'aumentare delle epoches. Quando si presenta questa problematica, è necessario determinarne le cause, le quali possono essere molteplici:

- Un possibile valore del *learning rate* troppo alto/basso;
- *Weight decay* troppo alto/basso. Questo parametro influenza i pesi facendoli decadere verso 0 se non ci sono più istanze per aggiornarli;
- La funzione di *data augmentation* utilizzata produce immagini poco significative (immagini poco diverse dalle originali). Viceversa una *data augmentation* troppo complessa rallenta di molto l'addestramento.
- E' necessario modificare le dimensioni dell'immagine, 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*).

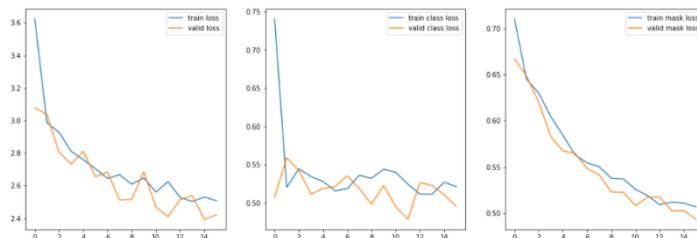
La configurazione iniziale di Mask R-CNN è stata la seguente:

```

1 class FashionConfig(Config):
2     NAME = "fashion"
3     NUM_CLASSES = 46 + 1 # +1 for the background class
4     GPU_COUNT = 1
5     IMAGES_PER_GPU = 4
6     BACKBONE = 'resnet50'
7     IMAGE_MIN_DIM = 1024
8     IMAGE_MAX_DIM = 1024
9     IMAGE_RESIZE_MODE = 'none'
10    DATA_AUGMENTATION = 'none'
11    RPN_ANCHOR_SCALES = (16, 32, 64, 128, 256)
12    STEPS_PER_EPOCH = 100
13    VALIDATION_STEPS = 50

```

La prima fase di training è stata eseguita con questi parametri per lo studio del loro impatto sul modello finale. Sono state eseguite 30 epoche di training, ciascuna avente 100 steps. I risultati ottenuti dopo 20 epoche sono i seguenti:



Dal grafico si può notare che l'epoca migliore è stata la 15^a con una *valid loss* di 2.394117850065231.

Siccome sono stati utilizzati i pesi già addestrati in precedenza sul MS COCO

database, è stato importante verificare più volte se si ottengessero risultati migliori durante il training del modello aggiornando solamente i pesi relativi agli ultimi *layers* della rete. Quindi aggiornando solamente i *layers head*, ovvero i *fully-connected layers*, e "congelando" i pesi relativi agli ulteriori *layers*. La scelta è ricaduta su soluzioni ibride per cercare il migliore compromesso di tempo e precisione tramite:

- variazione del *learning rate* ogni n epoche, di solito diminuendolo.
- variazione dei *layers* coinvolti nel processo di aggiornamento dei pesi
- variazione del *learning rate* + variazione dei *layers*

Al fine di velocizzare il processo di training e ottenere una maggiore precisione, sono state effettuate due modifiche principali ai parametri del modello:

```

1 class FashionConfig(Config):
2     NAME = "fashion"
3     NUM_CLASSES = 46 + 1 # +1 for the background class
4     GPU_COUNT = 1
5     IMAGES_PER_GPU = 4
6     BACKBONE = 'resnet101'
7     IMAGE_MIN_DIM = 512
8     IMAGE_MAX_DIM = 512
9     IMAGE_RESIZE_MODE = 'none'
10    DATA_AUGMENTATION = 'none'
11    RPN_ANCHOR_SCALES = (16, 32, 64, 128, 256)
12    STEPS_PER_EPOCH = 100
13    VALIDATION_STEPS = 50

```

La prima differenza è la dimensione dell'immagini impostata a 512×512 , che ha portato ad un dimezzamento del tempo per l'addestramento senza avere ripercussioni sulla precisione. La seconda differenza è stata la scelta di *resnet101* come funzione di *backbone*. L'utilizzo di *resnet101* al posto di *resnet50* ha aumentato la velocità di training, mantenendo una *validation loss* stabile o in diminuzione.

5.2.1 Data augmentation

Come è stato approfondito nel capitolo 4 una rete neurale della famiglia CNN è in grado di classificare con una buona precisione oggetti anche se orientati in diversi modi, siccome le reti convoluzionali possiedono la proprietà di invarianza. Più specificamente, una CNN può essere invariante alla traslazione, al cambiamento di punto di vista, dimensione o illuminazione (ma anche ad una combinazione di questi). Perciò la tecnica di *data augmentation* consiste proprio nel manipolare i dati aggiungendo particolari effetti al fine di fornire

contesti diversi alla rete. 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.

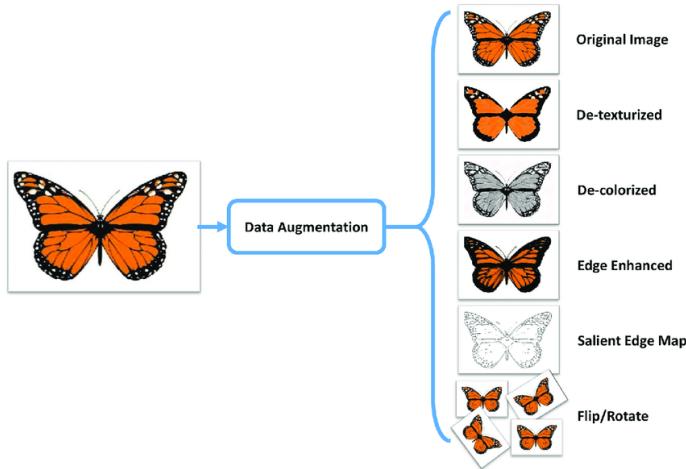


Figura 5.2: Data augmentation su una istanza di farfalla

Il risultato è un modello che riesce ad estrapolare meglio il contenuto semantico delle immagini, evitando di apprendere *pattern* irrilevanti ed in sostanza aumentando le prestazioni complessive.

Bisogna però porre attenzione a quale tipo di manipolazione si sottopongono i dati, valutando il più possibile il compito di classificazione proposto. Supponendo di avere a che fare con un problema di riconoscimento semaforico da parte di un agente, diviene inutile applicare un effetto di *data augmentation* che modifichi il colore del semaforo, siccome questo tipo di effetto produrrà comportamenti errati.

Il dataset fornito in input possedeva solamente immagini in orizzontale e verticale, ma lo scenario in cui potrebbe operare potrebbe essere molto più vario. Nonostante il grande numero di immagini su cui addestrare la rete neurale, la tecnica di *data augmentation* è stata utilizzata anche per cercare di evitare l'*overfitting* del modello. La funzione utilizzata durante il tirocinio è ottenuta dalla composizione di più pratiche:

```

1 # Image augmentation
2 augmentation = iaa.Sequential([
3     iaa.OneOf([
4         iaa.Affine(rotate=0),
5         iaa.Affine(rotate=90),
6         iaa.Affine(rotate=180),
  
```

```

7     iaa.Affine(rotate=270),
8 ],
9 iaa.Fliplr(0.5),
10 iaa.Flipud(0.5),
11 iaa.OneOf([
12     ## brightness or contrast
13     iaa.Multiply((0.9, 1.1)),
14     iaa.ContrastNormalization((0.9, 1.1)),
15 ]),
16 iaa.OneOf([
17     ## blur or sharpen
18     iaa.GaussianBlur(sigma=(0.0, 0.3)),
19     iaa.Sharpen(alpha=(0.0, 0.3)),
19 ]),
19 ])

```

Il risultato in output:



5.2.2 k-Fold Cross-Validation

k-Fold Cross-Validation è una procedura utilizzata per valutare l'apprendimento dei modelli su un campione di dati, perciò stimare l'abilità di fare previsioni su dati al di fuori del dataset d'addestramento e di test. La funzione ha un singolo parametro k , che si riferisce al numero di gruppi in cui deve essere suddiviso il dataset.

È un metodo molto usato per la semplicità di comprensione e implementazione, inoltre restituisce una stima meno distorta e meno ottimistica dell'abilità del modello rispetto ad altri metodi. La procedura generale prevede i seguenti passi:

1. Il dataset viene diviso in k -gruppi tramite una scelta casuale.

2. Per ogni gruppo K partendo da $i = 0$ (primo gruppo):
 - (a) Tutti i gruppi escluso l' i -esimo vengono utilizzati come set dati di addestramento .
 - (b) il gruppo i -esimo sarà utilizzato come un blocco di test per la valutazione del modello.

L'output rappresenterà il punteggio di valutazione dei diversi modelli.

Questa tecnica è stata implementata nel seguente modo:

```

1 FOLD = 0
2 N_FOLDS = 5
3
4 kf = KFold(n_splits=N_FOLDS, random_state=42, shuffle=True)
5 splits = kf.split(image_df) # ideally, this should be
   multilabel stratification
6
7 def get_fold():
8     for i, (train_index, valid_index) in enumerate(splits):
9         if i == FOLD:
10             return image_df.iloc[train_index], image_df.iloc[
11                 valid_index]
12
13 train_df, valid_df = get_fold()
14
15 train_dataset = FashionDataset(train_df)
16 train_dataset.prepare()
17
18 valid_dataset = FashionDataset(valid_df)
19 valid_dataset.prepare()
```

Ha portato notevoli miglioramenti e soprattutto un controllo costante sulle prestazioni.

5.2.3 Controllo delle prestazioni tramite mAP

Il controllo della precisione del modello è stato fatto durante la stessa fase d'addestramento sfruttando *k-Fold Cross-Validation* e il calcolo del *mean Average Precision*, quest'ultima approfondita nella sezione 4.5.3.

Per effettuare il calcolo durante il training è stata utilizzata la funzione di *callbacks* di *TensorFlow*³.

Di seguito la funzione di mAP utilizzata:

³piattaforma open source end-to-end per il machine learning. Ha un ecosistema completo e flessibile di strumenti, librerie e risorse della comunità per il ML

```

1 class MeanAveragePrecisionCallback(Callback):
2     def __init__(self, train_model: MaskRCNN, inference_model
3      : MaskRCNN, dataset: Dataset,
4      calculate_map_at_every_X_epoch=5,
5      dataset_limit=None, verbose=1):
6         super().__init__()
7         self.train_model = train_model
8         self.inference_model = inference_model
9         self.dataset = dataset
10        self.calculate_map_at_every_X_epoch =
11            calculate_map_at_every_X_epoch
12        self.dataset_limit = len(self.dataset.image_ids)
13        if dataset_limit is not None:
14            self.dataset_limit = dataset_limit
15        self.dataset_image_ids = self.dataset.image_ids.copy()
16
17        if inference_model.config.BATCH_SIZE != 1:
18            raise ValueError("This callback only works with
19            the batch size of 1")
20
21        self._verbose_print = print if verbose > 0 else
22            lambda *a, **k: None
23
24    def on_epoch_end(self, epoch, logs=None):
25
26        if epoch > 2 and (epoch+1)%self.
27        calculate_map_at_every_X_epoch == 0:
28            self._verbose_print("Calculating mAP...")
29            self._load_weights_for_model()
30
31            mAPs = self._calculate_mean_average_precision()
32            mAP = np.mean(mAPs)
33
34            if logs is not None:
35                logs["val_mean_average_precision"] = mAP
36
37            self._verbose_print("mAP at epoch {0} is: {1}".
38            format(epoch+1, mAP))
39
40            super().on_epoch_end(epoch, logs)
41
42    def _load_weights_for_model(self):
43        last_weights_path = self.train_model.find_last()
44        self._verbose_print("Loaded weights for the inference
45        model (last checkpoint of the train model): {0}".
46            format(
47                last_weights_path))
48        self.inference_model.load_weights(last_weights_path,
49                                         by_name=True)

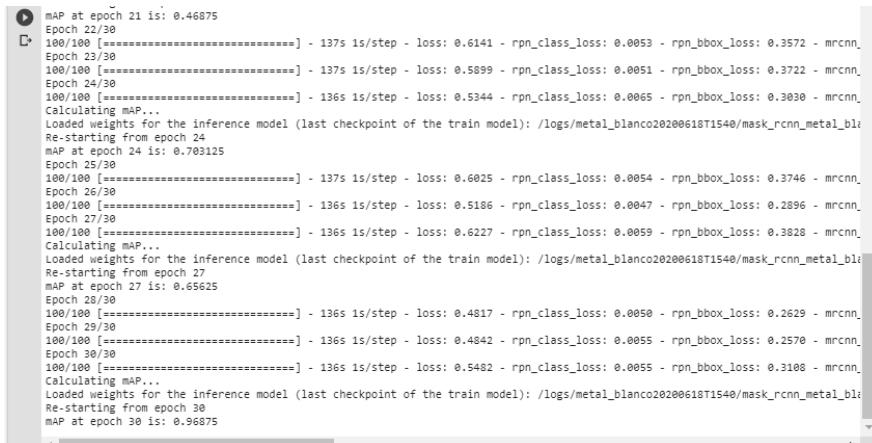
```

```

40
41     def _calculate_mean_average_precision(self):
42         mAPs = []
43
44         # Use a random subset of the data when a limit is
45         # defined
46         np.random.shuffle(self.dataset_image_ids)
47
48         for image_id in self.dataset_image_ids[:self.
49             dataset_limit]:
50             image, image_meta, gt_class_id, gt_bbox, gt_mask
51             = load_image_gt(self.dataset, self.inference_model.config,
52
53                 image_id, use_mini_mask=False)
54             molded_images = np.expand_dims(mold_image(image,
55                 self.inference_model.config), 0)
56             results = self.inference_model.detect(
57                 molded_images, verbose=0)
58             r = results[0]
59             # Compute mAP - VOC uses IoU 0.5
60             AP, _, _, _ = utils.compute_ap(gt_bbox,
61                 gt_class_id, gt_mask, r["rois"],
62                                         r["class_ids"], r[
63                 "scores"], r['masks'])
64             mAPs.append(AP)
65
66
67         return np.array(mAPs)

```

La funzione è stata implementata come un *custom callback*, perciò veniva utilizzata al termine di ogni X epoche in base al valore di calculate_map_at_every_X_epoch. La metrifica ha permesso un controllo della precisione tale che durante l'inferenza del modello non si sono riscontrate perdite significative di precisione.



The screenshot shows a terminal window with the following output:

```

mAP at epoch 21 is: 0.46875
Epoch 22/30
Epoch 23/30
100/100 [=====] - 1375 1s/step - loss: 0.6141 - rpn_class_loss: 0.0053 - rpn_bbox_loss: 0.3572 - mrcnn_
Epoch 24/30
100/100 [=====] - 1375 1s/step - loss: 0.5899 - rpn_class_loss: 0.0051 - rpn_bbox_loss: 0.3722 - mrcnn_
Epoch 25/30
100/100 [=====] - 1365 1s/step - loss: 0.5344 - rpn_class_loss: 0.0065 - rpn_bbox_loss: 0.3030 - mrcnn_
Calculating mAP...
Loaded weights for the inference model (last checkpoint of the train model): /logs/metal_blanco20200618T1540/mask_rcnn_metal_bla
Re-starting from epoch 24
mAP at epoch 24 is: 0.709125
Epoch 25/30
100/100 [=====] - 1375 1s/step - loss: 0.6025 - rpn_class_loss: 0.0054 - rpn_bbox_loss: 0.3746 - mrcnn_
Epoch 26/30
100/100 [=====] - 1365 1s/step - loss: 0.5186 - rpn_class_loss: 0.0047 - rpn_bbox_loss: 0.2896 - mrcnn_
Epoch 27/30
100/100 [=====] - 1365 1s/step - loss: 0.6227 - rpn_class_loss: 0.0059 - rpn_bbox_loss: 0.3828 - mrcnn_
Calculating mAP...
Loaded weights for the inference model (last checkpoint of the train model): /logs/metal_blanco20200618T1540/mask_rcnn_metal_bla
Re-starting from epoch 27
mAP at epoch 27 is: 0.65625
Epoch 28/30
100/100 [=====] - 1365 1s/step - loss: 0.4817 - rpn_class_loss: 0.0050 - rpn_bbox_loss: 0.2629 - mrcnn_
Epoch 29/30
100/100 [=====] - 1365 1s/step - loss: 0.4842 - rpn_class_loss: 0.0055 - rpn_bbox_loss: 0.2570 - mrcnn_
Epoch 30/30
100/100 [=====] - 1365 1s/step - loss: 0.5482 - rpn_class_loss: 0.0055 - rpn_bbox_loss: 0.3108 - mrcnn_
Calculating mAP...
Loaded weights for the inference model (last checkpoint of the train model): /logs/metal_blanco20200618T1540/mask_rcnn_metal_bla
Re-starting from epoch 30
mAP at epoch 30 is: 0.96875

```

Dai dati precedenti si può vedere come nella 30^a epoca si sia raggiunto un livello di confidenza⁴ medio di 0.96875 su 1.

5.3 Inferenza del modello

Ottenuto un modello sufficientemente prestazionale il passo successivo è l'inferenza.

Per inferenza si intende il processo di valutazione di nuove immagini utilizzando il modello addestrato per ottenere una risposta. L'input per questo compito sono state immagini che non avevano associato alcuna label e l'output ottenuto sarà l'immagine con sovrapposte le maschere, con associate il loro livello di confidenza.

L'implementazione di questa fase è avvenuta tramite il seguente codice:

```
1 class InferenceConfig(FashionConfig):
2     GPU_COUNT = 1
3     IMAGES_PER_GPU = 1
4
5 inference_config = InferenceConfig()
6
7 model = modellib.MaskRCNN(mode='inference', config=
8     inference_config, model_dir=ROOT_DIR)
9 model.load_weights(model_path, by_name=True)
10
11 for i in range(9):
12     image_id = train_df.sample()['ImageId'].values[0]
13     image_path = str(DATA_DIR/'image_c'/image_id)
14     img = cv2.imread(image_path)
15     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
16     result = model.detect([resize_image(image_path)])
17     r = result[0]
18     if r['masks'].size > 0:
19         masks = np.zeros((img.shape[0], img.shape[1], r['
20         masks'].shape[-1]), dtype=np.uint8)
21         for m in range(r['masks'].shape[-1]):
22             masks[:, :, m] = cv2.resize(r['masks'][:, :, m],
23                                         (img.shape[1], img.shape[0]),
24                                         interpolation=cv2.INTER_NEAREST)
25
26         y_scale = img.shape[0]/IMAGE_SIZE
27         x_scale = img.shape[1]/IMAGE_SIZE
28         rois = (r['rois'] * [y_scale, x_scale, y_scale,
29         x_scale]).astype(int)
```

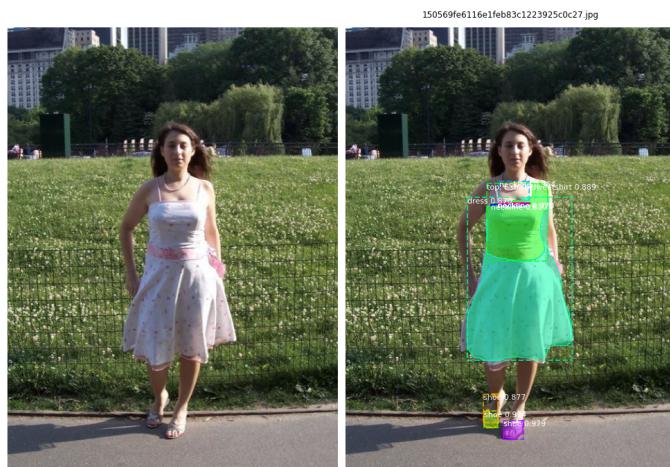
⁴è una misura di quanto il modello sia sicuro che l'istanza appartenga alla classe

```

26     masks, rois = refine_masks(masks, rois)
27 else:
28     masks, rois = r['masks'], r['rois']
29
30 visualize.display_instances(img, rois, masks, r['
class_ids'], ['bg']+label_names, r['scores'], title=
image_id, figsize=(12,12))

```

Il tirocinio si è concluso con la visione delle predizioni del modello, mostrate di seguito:



Capitolo 6

Conclusioni

Alla luce dei risultati presentati nel Capitolo 5, si può dire di aver modellato una rete neurale Mask R-CNN per il compito proposto. Di fatti i modelli ottenuti hanno una buona capacità di classificazione e segmentazione delle istanze contenute nelle immagini. Ovviamente nelle immagini dove la presenza di un numero di oggetti è elevata, maggiore è la probabilità che alcuni di questi siano sovrapposti l'uno con l'altro, rendendo complesso l'individuazione dell'istanze.

Il 30° modello, il migliore, si potrebbe utilizzare efficacemente per l'*instance segmentation* in real-time, convertendolo tramite librerie specifiche di *Python* in un linguaggio più performante come può essere *C++*. Inoltre si potrebbero aggiungere delle valutazioni delle prestazioni durante l'inferenza, come le metriche di COCO oppure una matrice di confusione.

In conclusione essendo una tecnologia nuova, in futuro sarà possibile sviluppare un modello per lo stesso compito con una complessità temporale minore e con prestazioni migliori.

Bibliografia

- [1] *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way / by Sumit Saha / Towards Data Science.* URL: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53>.
- [2] *A Gentle Introduction to k-fold Cross-Validation.* URL: <https://machinelearningmastery.com/k-fold-cross-validation/>.
- [3] *Backpropagation / Brilliant Math & Science Wiki.* URL: <https://brilliant.org/wiki/backpropagation/>.
- [4] Ayoub Benali Amjoud e Mustapha Amrouch. “Convolutional Neural Networks Backbones for Object Detection”. In: *Image and Signal Processing*. A cura di Abderrahim El Moataz et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 282–289. ISBN: 978-3-030-51935-3. DOI: [10.1007/978-3-030-51935-3_30](https://doi.org/10.1007/978-3-030-51935-3_30).
- [5] Jason Brownlee. *A Gentle Introduction to Pooling Layers for Convolutional Neural Networks.* Machine Learning Mastery. 21 Apr. 2019. URL: <https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/>.
- [6] Jason Brownlee. *Gradient Descent For Machine Learning.* Machine Learning Mastery. 22 Mar. 2016. URL: <https://machinelearningmastery.com/gradient-descent-for-machine-learning/>.
- [7] *Classification: Precision and Recall / Machine Learning Crash Course.* URL: <https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall>.
- [8] *Convolution - an overview / ScienceDirect Topics.* URL: <https://www.sciencedirect.com/topics/mathematics/convolution>.
- [9] *CS231n Convolutional Neural Networks for Visual Recognition.* URL: <https://cs231n.github.io/convolutional-networks/>.

- [10] Alexandra Deis. *Data Augmentation for Deep Learning*. Medium. 10 Ago. 2019. URL: <https://towardsdatascience.com/data-augmentation-for-deep-learning-4fe21d1a4eb9>.
- [11] *Fully Connected Layers in Convolutional Neural Networks: The Complete Guide*. MissingLink.ai. URL: <https://missinglink.ai/guides/convolutional-neural-networks/fully-connected-layers-convolutional-neural-networks-complete-guide/>.
- [12] *Gradient Descent — ML Glossary documentation*. URL: https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html.
- [13] *Gradient Descent Explained. A comprehensive guide to Gradient... / by Daksh Trehan / Towards Data Science*. URL: <https://towardsdatascience.com/gradient-descent-explained-9b953fc0d2c>.
- [14] *How Do Convolutional Layers Work in Deep Learning Neural Networks?* URL: <https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/>.
- [15] *iMaterialist (Fashion) 2019 at FGVC6*. URL: <https://kaggle.com/c/imaterialist-fashion-2019-FGVC6>.
- [16] *Intersection over Union (IoU) for object detection - PyImageSearch*. URL: <https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/>.
- [17] *K-Fold Cross Validation. Evaluating a Machine Learning model can... / by Krishni / Data Driven Investor / Medium*. URL: <https://medium.com/datadriveninvestor/k-fold-cross-validation-6b8518070833>.
- [18] Nitin Kumar Kain. *Understanding of Multilayer perceptron (MLP)*. Medium. 2 Dic. 2019. URL: https://medium.com/@AI_with_Kain/understanding-of-multilayer-perceptron-mlp-8f179c4a135f.
- [19] Simeon Kostadinov. *Understanding Backpropagation Algorithm*. Medium. 12 Ago. 2019. URL: <https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd>.
- [20] Z² Little. *Multi-Layer Perceptron (MLP)*. Medium. 1 Gen. 2020. URL: <https://xzz201920.medium.com/multi-layer-perceptron-mlp-4e5c020fd28a>.
- [21] *machine learning - How do backbone and head architecture work in Mask R-CNN? Cross Validated*. URL: <https://stats.stackexchange.com/questions/397767/how-do-backbone-and-head-architecture-work-in-mask-r-cnn>.

- [22] *machine-learning - Funzioni di attivazione / machine-learning Tutorial*. URL: <https://riptutorial.com/it/machine-learning/example/31624/funzioni-di-attivazione>.
- [23] “Mask rcnn”. Tesi di dott.
- [24] Michael A. Nielsen. “Neural Networks and Deep Learning”. In: (2015). Publisher: Determination Press. URL: <http://neuralnetworksanddeeplearning.com>.
- [25] *Papers with Code - R-CNN Explained*. URL: <https://paperswithcode.com/method/r-cnn>.
- [26] *Perceptron - an overview / ScienceDirect Topics*. URL: <https://www.sciencedirect.com/topics/engineering/perceptron>.
- [27] *Perceptrons & Multi-Layer Perceptrons: the Artificial Neuron - MissingLink*. MissingLink.ai. URL: <https://missinglink.ai/guides/neural-network-concepts/perceptrons-and-multi-layer-perceptrons-the-artificial-neuron-at-the-core-of-deep-learning/>.
- [28] Paolo Piacenti. *Le 4 funzioni di attivazione neuronale più usate negli Artificial Neural Network*. Medium. 18 Ott. 2019. URL: <https://medium.com/@paolopiacenti1/le-4-funzioni-di-attivazione-neuronale-pi%C3%B9-usate-negli-artificial-neural-network-41096953b85c>.
- [29] *Precision vs. Recall - An Intuitive Guide for Every Machine Learning Person - Analytics Vidhya*. URL: <https://www.analyticsvidhya.com/blog/2020/09/precision-recall-machine-learning/>.
- [30] *R-CNN (Object Detection). A beginners guide to one of the most... / by Sharif Elfouly / Medium*. URL: <https://medium.com/@selfouly/r-cnn-3a9beddf55a>.
- [31] *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms / by Rohith Gandhi / Towards Data Science*. URL: <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e>.
- [32] Madhu Ramiah. *Convolution Neural Network- The backbone of image classification*. Medium. 20 Giu. 2019. URL: <https://medium.com/@madhuramiah/convolution-neural-network-the-backbone-of-image-classification-ddd2a49a6efa>.
- [33] *ROI pooling vs. ROI align. In computer vision there are many... / by Firiuzza / Medium*. URL: <https://medium.com/@Firiuzza/roi-pooling-vs-roi-align-65293ab741db>.

- [34] *RoIAlign Explained / Papers With Code*. URL: <https://paperswithcode.com/method/roi-align>.
- [35] Oleksii Sheremet. *Intersection over union (IoU) calculation for evaluating an image segmentation model*. Medium. 7 Set. 2020. URL: <https://towardsdatascience.com/intersection-over-union-iou-calculation-for-evaluating-an-image-segmentation-model-8b22e2e84686>.
- [36] *Training Mask R-CNN to be a Fashionista (LB0.07)*. URL: <https://kaggle.com/adityamehndiratta/training-mask-r-cnn-to-be-a-fashionista-lb-0-07>.
- [37] *Understanding the Mathematics behind Gradient Descent. / by Parul Pandey / Towards Data Science*. URL: <https://towardsdatascience.com/understanding-the-mathematics-behind-gradient-descent-dde5dc9be06e>.
- [38] *What is Data Augmentation & how it works?* URL: <https://www.mygreatlearning.com/blog/understanding-data-augmentation/>.
- [39] *Why and how to Cross Validate a Model? / by Sanjay.M / Towards Data Science*. URL: <https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f>.

Ringraziamenti

I miei più sentiti ringraziamenti vanno al mio tutor aziendale Giorgio Gemignani. Una figura sempre presente e molto paziente, che a fatto sorgere in me l'interesse per il campo della computer vision.

I secondi ringraziamenti vanno al mio compagno Matteo Brunello, con cui ho condiviso intensamente questo percorso.

Grazie anche a chi mi ha accompagnato e sostenuto in questi tre anni.

Dichiarazione d'originalità

Io sottoscritto/a Edoardo Chiavazza nato a Torino il 03/05/1996 dichiaro che l'elaborato è frutto del mio lavoro originale, che nessuno lo ha scritto in mia vece, che non ho copiato il lavoro di altri e che ho documentato tutte le fonti che ho utilizzato.

Dichiaro anche che ho personalmente consultato tutte le fonti citate.

Dichiaro di non aver presentato questo elaborato presso altre istituzioni al fine di ottenere diplomi, lauree, certificazioni, ecc., né di averlo pubblicato in precedenza, in parte o per intero.

Dichiaro di aver letto e compreso che il ‘plagio’ è una “falsa attribuzione a sé di opere o scoperte delle quali spettino ad altri i diritti di invenzione o di proprietà” (Devoto-Oli, Dizionario della Lingua italiana, Milano, Le Monnier, 2001).

Dichiaro di aver compreso che quando si elabora un lavoro che incorpori parole o idee di altri, si deve citare appropriatamente la fonte di quell’informazione.

Dichiaro di dare credito all’autore sulle immagini sotto copyright e di non lucrare in alcun modo sull’elaborato, il quale realizzato solo per scopi didattici.