POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Sviluppo e test di applicazioni di Deep Learning su acceleratori hardware embedded

Relatore
Prof. Enrico Masala

Candidato
Fabio Ricciardi

Anno Accademico 2019/2020

Indice

Sommario

Al giorno d'oggi, è possibile osservare i prodotti dello sviluppo dell'Intelligenza Artificiale sia nella vita ordinaria che lavorativa. Finora, il progresso di tali tecniche ha sempre presupposto una macchina dotata di un'elevata capacità computazionale, ma, con lo sviluppo delle nuove tecnologie, si è affermata la tecnica di "*Inference at the Edge*", ovvero Inferenza alla periferia, che permette a dispositivi non capaci di sfruttare appieno le tecniche di Intelligenza Artificiale ed, in particolare, di Deep Learning, di eseguire la fase di inferenza in tempo reale senza alcun supporto esterno, se non dei dispositivi hardware embedded, garantendo privacy, consumi, costi ridotti ed esecuzione in tempo reale.

Lo scopo di questa tesi è lo sviluppo e il test di algoritmi e applicazioni di Machine Learning e Deep Learning eseguendo la fase d'inferenza, valutando le prestazioni, sull'Intel® Neural Compute Stick 2 (NCS2), dispositivo sviluppato da Intel® dalle dimensioni e forme di una chiavetta USB ma che all'interno nasconde un hardware capace di eseguire la fase di inferenza per un modello di rete neurale artificiale.

La tesi si articola in tre parti.

La prima parte propone un'introduzione al tema dell'Intelligenza Artificiale, spiegando i concetti di Machine Learning e Deep Learning e i loro fondamenti teorici e matematici in un approfondimento sulle reti neurali convoluzionali con un *focus* sul tema dell'Object Detection.

La seconda parte riguarda l'analisi della NCS2 e, in particolare, delle sue specifiche hardware e software, i passi necessari per la corretta installazione, l'analisi delle sue prestazioni rispetto a dispositivi analoghi e, infine, l'esposizione dello sviluppo di un'applicazione che segue il paradigma Client-Server, in cui un mittente richiede la classificazione di un'immagine e il destinatario, sfruttando la NCS2 e le reti a sua disposizione, esegue la fase di inferenza fornendo al mittente il risultato.

La terza parte, nucleo di questa tesi, riguarda l'utilizzo della NCS2 con reti neurali addestrate, tramite la tecnica del Transfer Learning, per il tema dell'Agricoltura di Precisione, in particolare per la viticoltura. Essa consiste nell'illustrazione dei software utilizzati e dei presupposti necessari al loro corretto funzionamento,

nonché dei risultati ottenuti addestrando dei modelli di reti neurali artificiali nell'identificazione di grappoli d'uva in una vigna e nel riconoscimento di foglie di vite malate. Infine, vi è una parte di analisi delle prestazioni dei modelli utilizzando la NCS2.

La tesi, divisa in sei capitoli, ha l'obiettivo di illustrare le competenze di base per il Machine Learning e il Deep Learning, ed in particolare per l'Object Detection, di illustrare la corretta installazione del dispositivo NCS2 e dimostrare come quest'ultimo sia utile per un tema attuale come l'agricoltura di precisione e che, magari, proseguendo il lavoro, si potrà prendere in considerazione lo sviluppo di ulteriori applicazioni con il supporto di dispositivi simili.

Capitolo 1

Introduzione

1.1 Panoramica

Al giorno d'oggi le tecniche di Intelligenza Artificiale, ovvero di "apprendimento automatico", sono presenti in molteplici strumenti della nostra vita, sia ordinaria che lavorativa. Esempi noti sono il riconoscimento del viso quando apriamo l'applicazione della fotocamera sul nostro smartphone oppure l'assistente vocale che risponde intelligentemente alle nostre domande e richieste.

Il più delle volte, tali tecniche presuppongono una macchina dotata di un'elevata capacità computazionale, presente fisicamente o nel cloud, anche se, negli ultimi anni, si è affermata la tecnica "*Inference at the Edge*", ovvero Inferenza alla periferia, che permette a dispositivi, i quali normalmente non sono capaci fisicamente di sfruttare appieno le reti neurali, di eseguire la fase di inferenza in tempo reale senza alcun supporto esterno se non degli acceleratori hardware embedded.

In questo modo, vengono fornite a dispositivi come videocamere di sorveglianza, droni o rover, le capacità per le quali la rete neurale è stata realizzata.

1.2 Scopo della Tesi

Lo scopo di questa tesi è lo sviluppo di applicazioni e il test di algoritmi di *Machine Learning* e di *Deep Learning* realizzando la fase di inferenza attraverso **Intel® Neural Compute Stick 2** (**NCS2**), dispositivo embedded dalle dimensioni e forme di una chiavetta USB capace di eseguire inferenza

alla periferia, così da dimostrare come dispositivi destinati all'*Internet of Things* e dalla bassa capacità di calcolo computazionale possano eseguire la fase di inferenza delle reti neurali, garantendo molteplici vantaggi di cui in seguito.

La prima parte propone un'introduzione al mondo dell'intelligenza artificiale, focalizzandosi sul Machine Learning e sul Deep Learning, e in particolare sull'*Object Detection*, spiegando le differenze tra le due branche e quali fasi vengono attraversate da una rete neurale prima del suo effettivo utilizzo sul campo.

La seconda parte, invece, si concentra sulle specifiche della NCS2, i suoi componenti hardware e software e sul benchmarking del dispositivo confrontandolo con il suo predecessore e alcuni analoghi dispositivi. Oltre a ciò, verrà spiegato il toolkit *OpenVINO™*, sviluppato anch'esso da Intel®, necessario al corretto funzionamento della NCS2.

Successivamente, viene mostrato lo sviluppo di un'applicazione scritta in linguaggio *Python*, basata sul paradigma Client-Server, in cui un mittente invia in rete una serie di comandi diretti a un destinatario, il quale fa uso della NCS2 per eseguire l'inferenza di alcuni modelli di deep learning e restituire il risultato al mittente.

La tesi prosegue introducendo il tema dell'agricoltura di precisione e mostra i risultati che vengono ottenuti sfruttando alcuni modelli di reti neurali addestrati per la localizzazione di grappoli d'uva, foglie di vite e classificazione di quest'ultime per quanto ne riguarda lo stato di salute.

Infine, vengono mostrate alcune applicazioni pratiche di utilizzo di tali reti sfruttando la NCS2, valutandone i risultati.

Capitolo 2

Machine Learning e reti neurali artificiali

2.1 Machine Learning e Deep Learning

Nella tesi si fa spesso riferimento a **Machine Learning [**1] e **Deep Learning** [2], termini accostati poiché sottoinsiemi dell'Intelligenza Artificiale ma che sono molto differenti tra loro.

Machine Learning è l'insieme delle tecniche volte ad allenare l'intelligenza artificiale perché possa svolgere attività non programmate, apprendendo dall'esperienza pregressa e migliorandosi attraverso gli errori commessi; tuttavia, nei modelli di Machine Learning il programmatore deve comunque guidare esplicitamente l'algoritmo in caso di previsione errata; problematica affrontata invece nel *Deep Learning*.

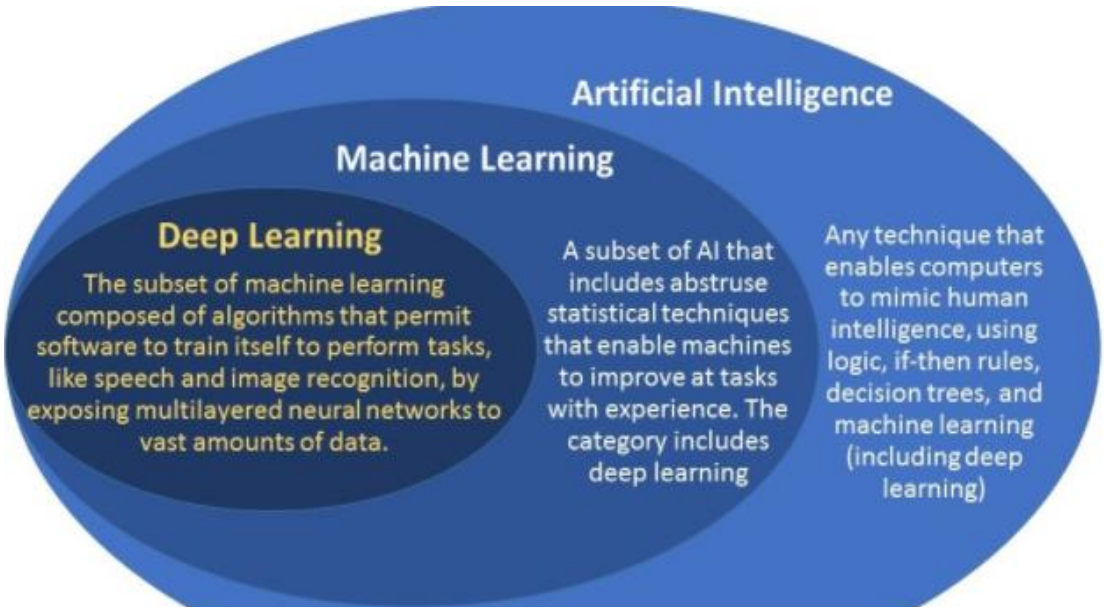


Figura 2.1 diagramma riassuntivo di Intelligenza Artificiale, Machine Learning e Deep Learning. Fonte Immagine e riferimenti: [3]

Deep Learning è un sottoinsieme del Machine Learning e funziona allo stesso modo ma con diverse capacità; infatti, un modello di Deep Learning cerca di simulare la mente umana correggendosi autonomamente in caso di errori. Il termine "*Deep*" deriva dal fatto che le reti neurali sono dotate di molteplici livelli nascosti di neuroni, rendendole, di fatto, profonde e somiglianti alla struttura del cervello umano.

Per quanto riguarda le tecniche di apprendimento, possiamo suddividere gli algoritmi di Machine Learning in due categorie:

- **Apprendimento Supervisionato**: "supervisionato" perché la presenza delle soluzioni, denominate anche "etichette", è fornita nell'insieme dei dati di addestramento dal programmatore, che pertanto prende il nome di "supervisore". Il supervisore fornisce all'algoritmo degli esempi di cui sono indicate le variabili di input con la previsione corretta e, attraverso tali esempi, l'algoritmo elabora un modello predittivo. Un esempio pratico è il filtro della casella e-mail per distinguere le mail di "spam" da quelle "no spam" [4];
- **Apprendimento Non Supervisionato**: in questo caso le etichette non vengono fornite: sarà l'algoritmo, per come è strutturato, a trovare una logica di classificazione. Esempi pratici si hanno negli attuali motori di ricerca, i quali, data una parola chiave, sono in grado di creare una lista di link rimandanti alle pagine che l'algoritmo di ricerca ritiene attinenti a quella effettuata [5].

Vi sarebbe, inoltre, una terza categoria denominata

- **Apprendimento per Rinforzo**: in questo caso non vi sono esempi di associazioni input-output, né tantomeno aggiustamenti espliciti degli output da ottimizzare in quanto l'algoritmo apprende esclusivamente interagendo con l'ambiente, ovvero attraverso una politica di incentivi, se l'azione compiuta è positiva, e disincentivi, in caso contrario. In questo modo, si cerca di addestrare un algoritmo a trovare soluzioni "creative"; un esempio è stata l'applicazione di tale apprendimento con il videogioco *Arcade Breakout*, in cui, dopo sole quattro ore, la rete trovò una strategia di approccio al gioco mai pensata prima da un essere umano. [6]

Nello svolgimento di questa tesi, l'algoritmo utilizzato è stato quello dell'Apprendimento Supervisionato.

2.2 Reti Neurali

Il miglior modo per capire il funzionamento di una rete neurale è capire come funziona un neurone biologico.

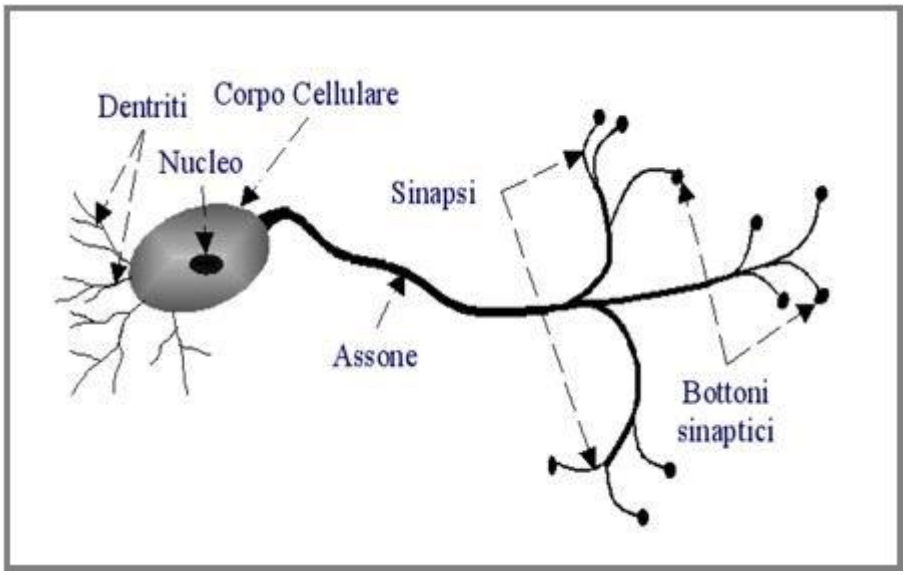


Figura 2.2 Rappresentazione schematica di un neurone biologico. Fonte immagine e riferimenti: [7].

I principali componenti di un neurone biologico (figura 2.2) sono:

- Soma: parte centrale del neurone;
- Assone: un filamento che fuoriesce dal soma;
- Sinapsi: "ramificazione" dell'assone; ciascuna sinapsi termina con un bottone sinaptico;
- Dendriti: filamenti che fuoriescono dal soma.

Il neurone è capace di ricevere segnali attraverso i propri dendriti, li elabora nel soma e successivamente trasmette il segnale, tramite l'assone, al neurone successivo. L'assone non è direttamente collegato ai dendriti di altri neuroni: il punto in cui il segnale viene trasmesso da una cellula ad un'altra è un piccolo spazio denominato "fessura sinaptica". Quando un segnale è nei pressi di una sinapsi, questa rilascia un quantitativo di sostanze chimiche

chiamate “neurotrasmettitori”, i quali determinano la conduttività di una sinapsi, ovvero quanto la sinapsi attenua o enfatizza il segnale elettrico dall’assone. Nella trasmissione di un segnale, le correnti si possono sommare in spazio e tempo e se tale somma oltrepassa una certa soglia, un impulso di una certa entità e durata, denominato “potenziale d’azione”, è generato. Il segnale così prosegue per il prossimo assone, ricominciando il processo. Le reti neurali si basano sulla simulazione di neuroni artificiali opportunamente collegati, i quali ricevono in ingresso degli stimoli elaborandoli di conseguenza. Il primo modello di neurone venne introdotto da Warren McCulloch e Walter Pitts nel 1943.

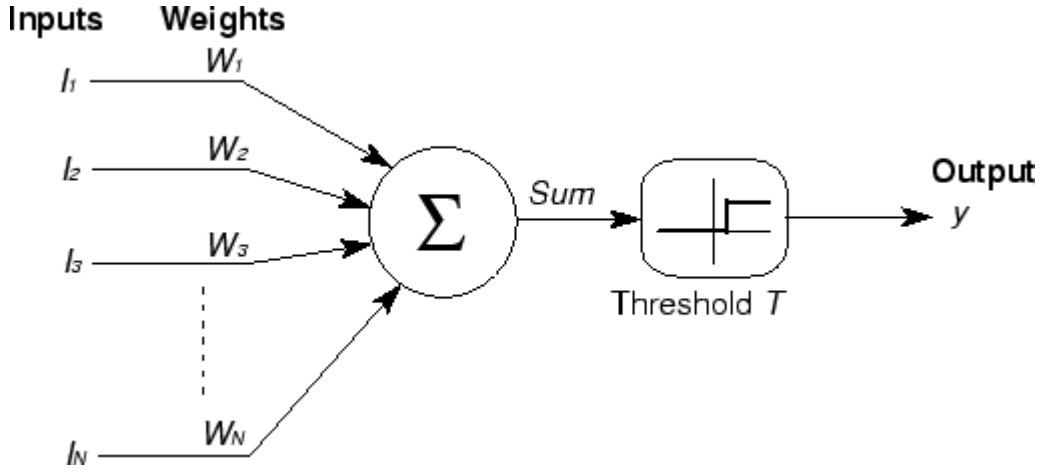


Figura 2.3 Rappresentazione del modello di McCulloch e Pitts. Fonte immagine e riferimenti: [8].

L’elaborazione prevede, nei casi più semplici, che i singoli ingressi vengano moltiplicati per un opportuno valore, detto “peso”, e il risultato delle moltiplicazioni venga poi sommato; se tale somma supera una certa soglia, il neurone attiva la propria uscita. La distribuzione dei valori dei pesi varia in base all’importanza dell’ingresso: un ingresso importante avrà un peso elevato, a differenza di uno meno importante che avrà un valore inferiore. Tuttavia, tale modello non si rivelò molto pratico, in quanto, per avere i valori desiderati, bisognava impostare manualmente pesi e connessioni.

2.2.1 Percettrone

Alla fine degli anni Cinquanta, Frank Rosenblatt introdusse una rete composta di unità perfezionate dal modello di McCulloch-Pitts, il *Percettrone*, ovvero l’unione del concetto del modello precedente con la

regola di Hebbian [9] per l'adattamento dei pesi. Inoltre, il modello del percettrone aggiungeva un ulteriore valore di input che rappresenta il *bias*.

Le principali differenze con il modello McColloch-Pitts sono:

1. I pesi e le soglie non sono tutti identici;
2. I pesi possono assumere valori positivi e negativi;
3. Non esiste una sinapsi inibitoria assoluta;
4. Vi è una regola d'apprendimento.

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

Se consideriamo w e x come vettori e b (bias) come l'opposto della soglia possiamo riscrivere l'equazione nel seguente modo:

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

dove a è detta "funzione di attivazione". Il bias può essere interpretato come una soglia che influenza ampiamente l'output dell'unità. La novità principale del modello di Rosenblatt è la sua capacità di modificare i propri pesi adattandoli al problema dato in modo tale da non dover creare un circuito a mano [10].

Tuttavia, nel 1969, Marvin Minsky e Symour A. Papert, nel loro libro "*Perceptrons: an introducion to computational geometry*" [11], misero in luce tutti i limiti delle reti a due strati basate sui percettroni e come queste non fossero capaci di poter risolvere problemi se non quelli caratterizzati da separabilità lineare delle soluzioni, dando così inizio ad un periodo che prese il nome di "Inverno delle IA".

Al fine di risolvere problemi più complessi, si cominciò a interconnettere gli input dei neuroni artificiali con gli output di altri neuroni artificiali, creando una rete neurale a più livelli, ovvero il **percettrone multistrato.**

2.2.2 Percettrone multistrato

Il percettrone multistrato (MLP) è una vera e propria rete neurale artificiale composta, come si evince dal nome, da più percettroni. Esso si compone di un livello di input, il quale riceve il segnale, e un livello di output, che esegue una previsione o prende una decisione per quanto concerne l'input e, tra questi due livelli, vi è un numero arbitrario di strati "nascosti", il vero motore computazionale della rete. Ciascun neurone di un livello è connesso a tutti i neuroni del livello precedente, per tale motivo una rete di questo tipo è anche detta **Fully Connected**.

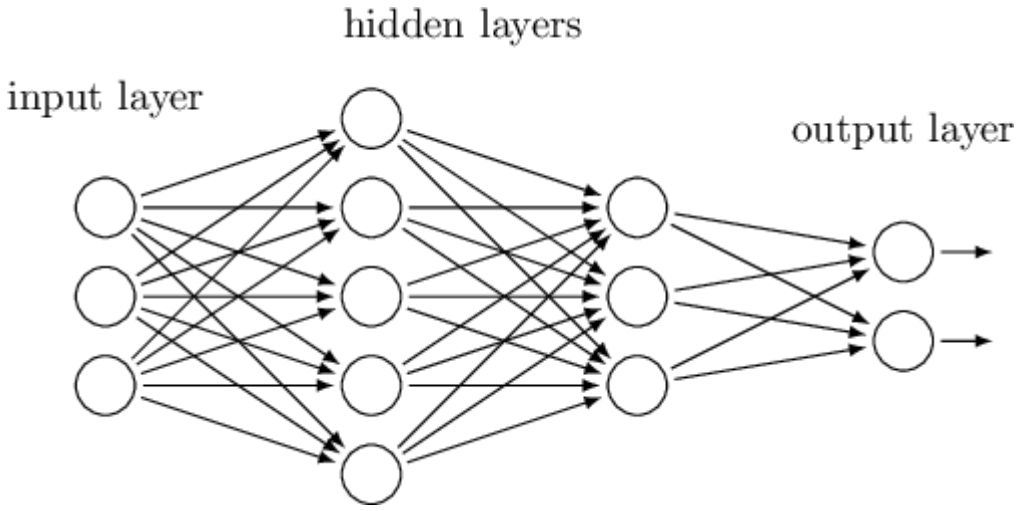


Figura 2.4 Esempio di MLP. Fonte immagine e riferimenti: [12].

2.3 Funzioni di attivazione

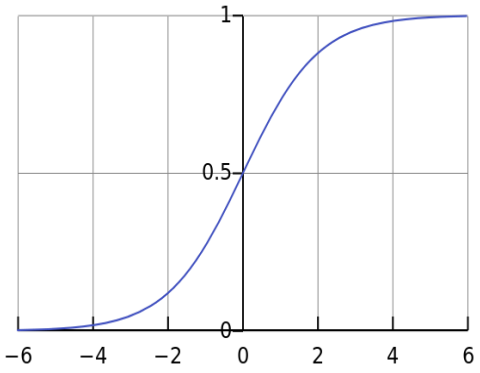
Al giorno d'oggi le reti neurali propongono vari tipi di funzioni di attivazione [13].

2.3.1 Sigmoide

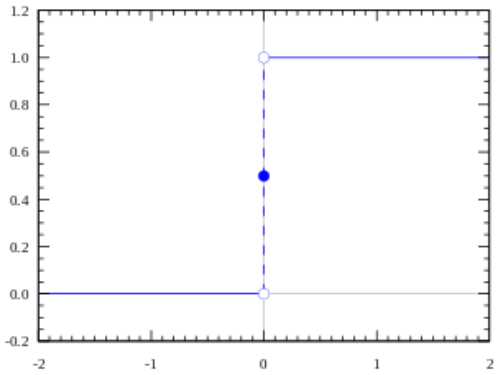
La differenza sostanziale di tale funzione rispetto alla formula (2.2) è la sua natura non lineare da cui si ottengono combinazioni non lineari.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

Di fatto, l'obiettivo della sigmoide è quello di ridurre gli effetti di piccole variazioni e di bilanciarli sull'output finale.



A) Grafico della sigmoide



B) Grafico della funzione gradino

La nuova funzione di attivazione diverrà dunque:

$$\sigma(wx + b) = \frac{1}{1 + \exp\left(-\sum_j w_j x_j - b\right)} \tag{2.4}$$

La sigmoide può essere vista come una versione smussata della funzione di attivazione del percettrone; tuttavia, anche se è una delle funzioni più usate oggi, non è esente da problemi: si noti come, nel grafico A), verso la fine della funzione, i valori delle coordinate tendano a rispondere molto meno rispetto alle ascisse. Questo fenomeno solleva il problema della *scomparsa del gradiente*, ovvero il gradiente ha assunto un valore talmente basso, quasi è scomparso, che la rete rifiuta di apprendere ulteriormente.

2.3.2 Tangente iperbolica

La funzione di tangente iperbolica (tanh) è una buona alternativa alla sigmoide. La sua natura è sempre non lineare ma il suo gradiente è molto più resistente della sigmoide e decidere tra le due dipenderà dalle richieste di robustezza del gradiente stesso; tuttavia, anche tale funzione non è esente dal problema della scomparsa del gradiente.

La sua equazione è:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.5}$$

Dunque, la funzione di attivazione diventa:

$$\sigma(z) = \frac{1 + \tanh(z/2)}{2}$$

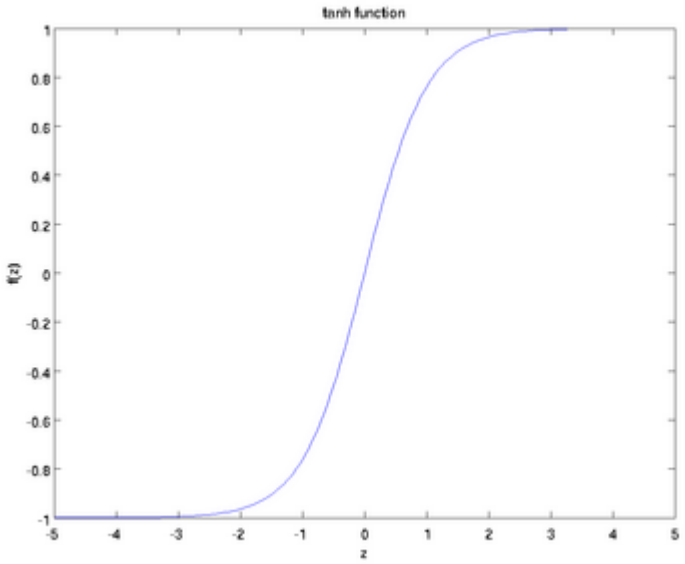


Figura 2.5 Grafico della funzione tangente iperbolica

2.3.3 SoftMax

Funzione di attivazione usata, il più delle volte, nell'output layer di una rete neurale, in particolare, nei problemi di classificazione [14]. Tale funzione accetta in input un vettore di K numeri reali normalizzandolo in una distribuzione di probabilità composta da K valori di probabilità sugli esponenziali dei valori in input. Ciò garantisce che, dopo averla applicata, i valori saranno sempre compresi nell'intervallo (0,1).

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \; per \; i = 1, \dots, K \; e \; z = (z_1, \dots, z_k) \in R^K \tag{2.6}$$

In pratica, si applica la funzione esponenziale per ogni elemento z_i del vettore input z e ogni valore viene normalizzato dividendolo per la somma di tutti gli esponenziali garantendo che la somma dei singoli componenti nel vettore output $\sigma(z)$ sia 1.

2.3.4 ReLU

Di questa funzione di attivazione parleremo in seguito nel paragrafo 2.5. Per ora si sappia che è una funzione dal carattere non lineare.

2.4 Addestramento di una rete neurale

Nel 1986 David E. Rumelhart, G. Hinton e R.J.Williams proposero il più conosciuto e utilizzato algoritmo per l'addestramento di una rete neurale: l'algoritmo della *retropropagazione dell'errore* (*Error Backpropagation*) [15]. Questo algoritmo consiste in una tecnica d'apprendimento tramite esempi, costituente una generalizzazione dell'algoritmo d'apprendimento del percettrone di Rosenblatt. L'algoritmo si basa sulla modifica sistematica dei pesi delle connessioni tra neuroni cosicché l'output della rete coincida sempre di più con il risultato aspettato. Distinguiamo due fasi principali nell'addestramento della rete: *forward propagation* e *backward propagation*.

Nella forward propagation i pesi assumono dei valori fissi (saranno dei valori di default alla prima iterazione) e vengono calcolate tutte le attivazioni dei neuroni della rete, dal primo layer proseguendo fino all'ultimo. Nella backward propagation, il risultato generato dalla rete viene confrontato con quello desiderato e se ne calcola l'errore. L'errore viene così propagato nel senso inverso a quello delle sinapsi, con l'intento di minimizzarlo, modificando i pesi di conseguenza. Alla fine di questa fase comincia una nuova iterazione con la forward propagation.

2.4.1 Forward Propagation

Definiamo *Forward Propagation* il processo di calcolo dell'output di una rete dati i suoi input. Consideriamo un problema di classificazione binaria utilizzando una rete composta da un singolo layer nascosto, un output layer con un solo neurone e la sigmoide come funzione di attivazione [16].

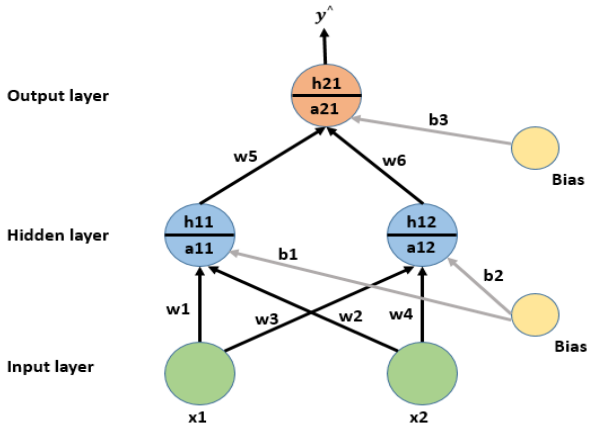


Figura 2.6 Rete Neurale d'esempio

Durante la forward propagation, ad ogni nodo del layer nascosto e dell'ouput layer avvengono le azioni di preattivazione e attivazione. Osserviamo la figura 2.6: ad esempio, nel primo livello vengono calcolati i valori di preattivazione $a_{11} e\ a_{12}$ i quali vengono passati alla relativa funzione di attivazione ottenendo $h_{11} e\ h_{12}$. Lo stesso discorso viene fatto sul nodo dell'output layer ottenendo così il valore di output predetto $\hat{y}$.

Definendo come $f_w(x_i)$ i valori di output stimati e come y_i le etichette reali, possiamo calcolare l'**errore** (o **perdita**) come:

$$E(w) = \sum_{i=1}^{N} \left(y_i - f_w(x_i)\right)^2 \tag{2.7}$$

Definiamo "**Training**" la fase di ricerca dei pesi che minimizzino l'errore quadratico dell'equazione (2.7). Data una funzione di attivazione *f* che sia differenziabile, la minimizzazione viene effettuata utilizzando il metodo della "*Discesa del gradiente*", tale fase è detta **Back Propagation.**

2.4.2 Back Propagation

Siano x_di valori di input, $t_d$le etichette, o_di valori di output osservati e w_il'i-esimo peso, si calcolano le derivate parziali dell'errore rispetto alle singole componenti:

$$\frac{\partial E(w)}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{2}\sum_d (t_d - o_d)^2 \tag{2.8}$$

Il gradiente $\nabla E(w)$ è l'insieme delle derivate rispetto a tutte le n componenti:

$$\nabla E(w) = [\frac{\partial E}{\partial w_o}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n}] \tag{2.9}$$

A questo punto si ottiene la "*training rule*" che definisce come cambiare i pesi delle connessioni con lo scopo di minimizzare l'errore.

$$\Delta w = -\mu \nabla E(w) = -\mu \frac{\partial E(w)}{\partial w} \tag{2.10}$$

Dove μ è definito "*learning rate*", un valore positivo ma molto piccolo, in modo da non eliminare le corrette classificazioni, e specifica il grado di apprendimento dei parametri. A questo punto i pesi vengono aggiornati con la seguente formula

$$w = w - \mu \nabla E(w)\,,$$

conosciuta, più comunemente, come **discesa del gradiente**.

L'applicazione di tale algoritmo a tutti i pesi delle sinapsi è definito **Retropropagazione dell'errore** e ha molteplici vantaggi:

- Esegue la discesa del gradiente a tutti i pesi dell'intera rete;
- È semplice da generalizzare per grafi diretti arbitrari;
- Riesce a trovare un minimo locale, non necessariamente uno globale.

Tuttavia:

- Il training potrebbe richiedere migliaia di operazioni.

2.5 Reti neurali convoluzionali

Le *reti neurali convoluzionali* (CNN) sono molto simili alle reti artificiali di cui si è parlato nel paragrafo precedente: sono composte da neuroni e hanno pesi e bias da apprendere. Ogni neurone ha degli input con i quali esegue un prodotto scalare e opzionalmente segue la non linearità; tuttavia, queste reti prevedono che l'input abbia una precisa struttura-dati, come, ad esempio, un'immagine, cosicché l'architettura sia vincolata in un certo modo.

La differenza fondamentale rispetto alle normali reti neurali artificiali è che gli strati di una CNN hanno dei neuroni disposti in tre dimensioni: **larghezza, altezza e profondità** (w,h,d); la profondità, in questo caso, indica la terza dimensione del volume di attivazione, non la profondità della rete neurale, come illustrato in precedenza. Inoltre, i neuroni di un livello saranno connessi a una piccola porzione del livello precedente, a differenza della strategia fully-connected, analizzata anch'essa in precedenza.

2.5.1 Architettura di una CNN

Nelle CNN distinguiamo tre tipi di livelli (*layer*): *Convolutional Layer*, *Pooling Layer* e *Fully-Connected Layer*. Le spiegazioni e le figure dei seguenti sotto-paragrafi sono tratti da un articolo dell'università di Stanford [17].

2.5.1.1 Convolutional Layer

Questi è il blocco portante di una CNN, in cui avvengono la maggior parte delle operazioni di computazione più pesanti. I suoi parametri consistono in un insieme di filtri da apprendere, ognuno molto piccolo dal punto di vista spaziale, per quanto riguarda larghezza e profondità, ma che si estende completamente lungo la dimensione della profondità dell'input. Ad esempio, un tipico filtro di un primo livello di una CNN potrebbe avere dimensioni 5x5x3 (i.e. il numero 5 sta ad indicare il numero di pixel per altezza e larghezza, mentre il numero 3 indica la profondità, poiché un'immagine ha 3 canali, RGB, per quanto riguarda i colori). Durante la forward propagation si convolve ciascun filtro lungo la larghezza e l'altezza del volume di input, producendo una *activation map* (o *feature map*) bidimensionale per quel filtro. La rete apprenderà, intuitivamente, i filtri che causano l'attivazione

dell'uscita del neurone quando osservano un certo tipo di tratto visuale, ad esempio una macchia di un certo colore sul primo layer. Seguendo questa procedura si finirà per ottenere un intero insieme di filtri per ogni Convolutional Layer, di cui ognuno produrrà una activation map bidimensionale, le quali verranno unite lungo la dimensione della profondità producendo il volume di output.

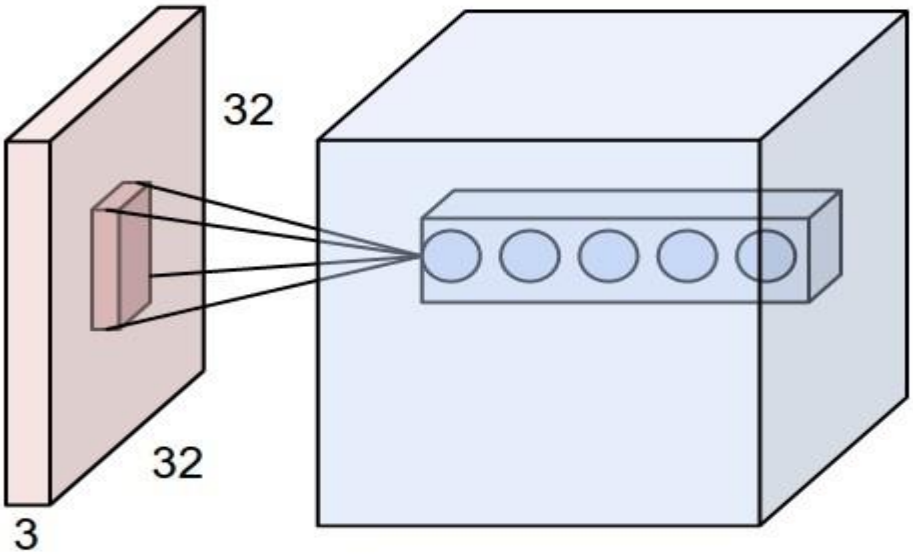


Figura 2.7 Esempio di convoluzione su immagine [32x32x3]

Facciamo un esempio pratico: supponiamo di avere un volume di input pari a [32x32x3] e la dimensione del filtro pari a [5x5]. Dunque, ogni neurone del Convolutional Layer avrà dei pesi per una regione del volume di input pari a [5x5x3] per un totale di75 pesi (+1 per il bias).

2.5.1.2 Organizzazione dello spazio

Quattro iperparametri controllano la dimensione del volume di ouput e vanno specificati:

1. La **profondità** (*depth)* del volume di output: corrisponde al numero di filtri che vogliamo usare, ognuno che apprende a osservare una particolarità differente dell'input. Definiamo l'insieme di neuroni che osservano la stessa regione dell'input come una *depth column;*

2. Il **passo** (*stride*) con cui il filtro trasla. Se il passo ha valore 1 allora il filtro trasla di un pixel alla volta. All'aumentare del passo, l'ouput avrà una dimensione spazialmente inferiore;
3. Delle volte conviene inserire degli zeri lungo il bordo dell'input, ciò che viene definito come *zero-padding*. Il valore dello zero-padding permette di avere sotto controllo la dimensione spaziale del volume di output;
4. Numero di filtri (*kernel*) da applicare.

Possiamo determinare la dimensione spaziale del volume di output come una funzione del valore del volume di input (**W**), la dimensione del filtro (**F**) dei neuroni nel Convolutional Layer, il passo che è stato applicato (**S**) e infine il valore di zero-padding usato sul bordo (**P**). La formula per calcolare quanti neuroni sono necessari è:

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

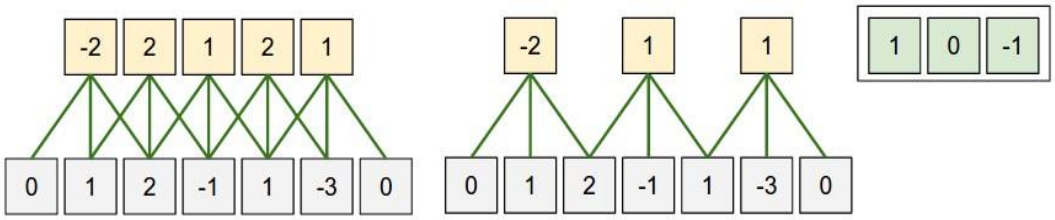


Figura 2.8 Illustrazione di organizzazione spaziale

Nell'esempio in figura 2.8 vi è una sola dimensione spaziale (asse x), un neurone con dimensione del filtro F = 3, dimensione di input W = 5 e vi è zero-padding P = 1. Guardando la figura da sinistra, nella prima immagine il neurone ha un passo S = 1, dando come output (5-3+2) /1 + 1 = 5. Nell'immagine seguente, il neurone ha un passo S = 2, restituendo come output (5-3+2) /2 + 1 = 3. Da notare come il passo S non possa assumere valore 3 in quanto non sarebbe compatibile con il volume. I pesi di questo neurone sono [-1,0,1] con bias nullo, e saranno condivisi attraverso i neuroni gialli.

2.5.1.3 Condivisione dei parametri

La condivisione dei parametri è usata nei Convolutional Layer per controllarne il numero. Prendiamo come esempio l'architettura che ha vinto la sfida di ImageNet, nel 2012, proposta da Krizhevsky A. *et al.*, la AlexNet [18]: vi sono 55x55x96 = 290.400 neuroni nel primo Convolutional Layer, ognuno dei quali ha 11x11x3 =363 pesi e 1 bias. Ciò porta a 290.400 * 364 = 105.705.600 parametri solo nel primo livello.

A quanto pare, però, si può ridurre drasticamente il numero di parametri con una sola ragionevole assunzione: se una feature è utile da calcolare in una precisa posizione (x,y) allora deve essere utile da calcolare anche in una posizione (x2,y2) differente.

In particolare, data una sezione bidimensionale lungo l'asse di profondità del volume di output nota come *depth slice* (nell'esempio precedente un volume di [55x55x96] ha 96 *depth slice*, ognuno di dimensione [55x55]), i neuroni di ogni sezione saranno vincolati ad usare gli stessi pesi e bias. Riprendendo l'esempio, avremo solo 96 set di pesi (uno per ogni sezione) per un totale di 96*11*11*3 = 34.848 pesi univoci, o 34.944 parametri (+96 bias). Durante la fase di back propagation, ogni neurone calcolerà il gradiente dei suoi pesi, ma tali gradienti saranno aggiunti attraverso i vari *depth slice* aggiornando un singolo insieme di pesi per ciascun *depth slice*.

Da notare che se tutti i neuroni in un singolo *depth slice* usano lo stesso vettore dei pesi, allora il passo di *forward propagation* del *Convolutional Layer*, in ogni *depth slice*, può essere calcolato come convoluzione dei pesi dei neuroni con il volume di input, da qui il nome **Convolutional Layer**. È per questo motivo che si è solito riferirsi agli insiemi dei pesi come un filtro (o kernel), il quale è convoluto con l'input.

2.5.1.4 ReLU layer

ReLU sta per "*Rectified Linear Unit*" ed è un tipo di funzione di attivazione. Dal punto di vista matematico si definisce come $y = \max(0, x)$ e visivamente ha il seguente aspetto:

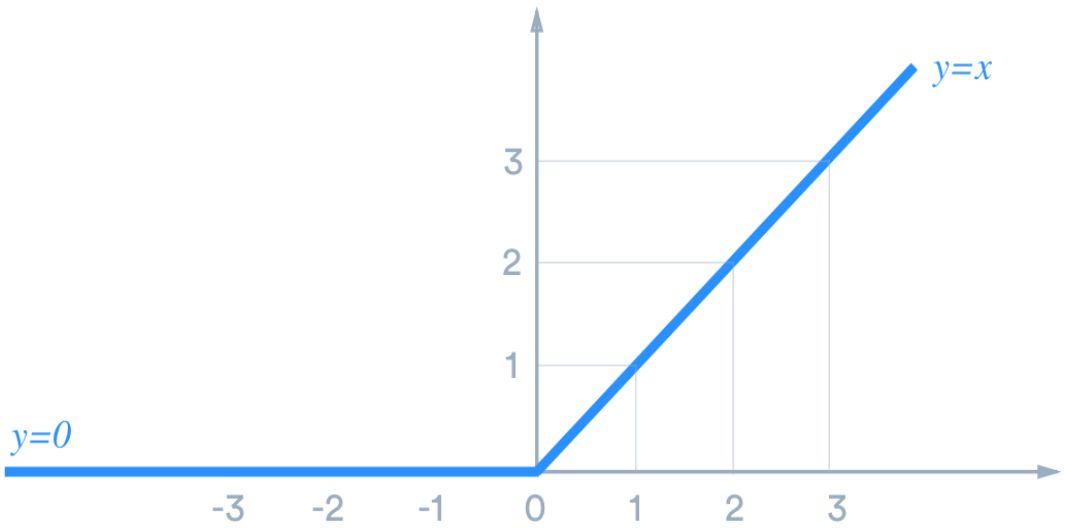


Figura 2.9 Diagramma della funzione ReLU. Fonte immagine e riferimenti: [19].

Essa è la funzione di attivazione più usata nelle reti neurali, specialmente nelle CNN, e assume un carattere lineare, per tutti i valori positivi, e nullo, per tutti quelli negativi. Ha molteplici vantaggi:

- Poco costosa dal punto di vista del calcolo in quanto non vi è una logica matematica complicata dietro;
- Converge più velocemente in quanto la linearità garantisce che la pendenza non "saturi" al crescere di x;
- Non soffre il problema della scomparsa del gradiente come per la sigmoide o la tangente iperbolica;
- Poiché la ReLU è nulla per gli input negativi, non ci sarà alcuna attivazione per tali tipi di input, comportamento spesso ricercato.

I ReLU layer si trovano dopo i Convolutional Layer e hanno la funzionalità di aumentare la proprietà di non linearità della funzione di attivazione senza modificare la dimensione del filtro.

2.5.1.5 Pooling Layer

È solito trovarsi, tra un *Convolutional Layer* e un altro, un layer intermedio chiamato *Pooling Layer*. La sua funzione principale è quella di ridurre progressivamente la dimensione spaziale della rappresentazione in modo tale da avere un numero inferiore di parametri e abbattere il costo di computazione della rete controllando il fenomeno del sovradattamento (o*verfitting*).

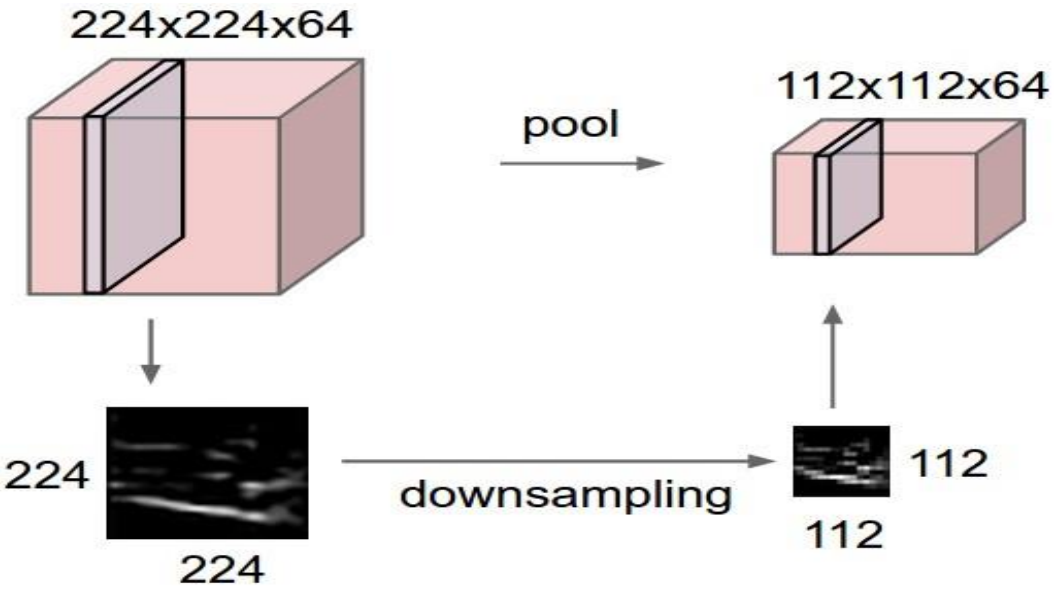


Figura 2.10 Esempio di operazione di pooling

Il *Pooling Layer* opera separatamente e indipendentemente su ogni *depth slice* del proprio input applicando un algoritmo di selezione dei parametri riducendo la dimensione spaziale. L'algoritmo più comune è quello del *MAX pooling*.

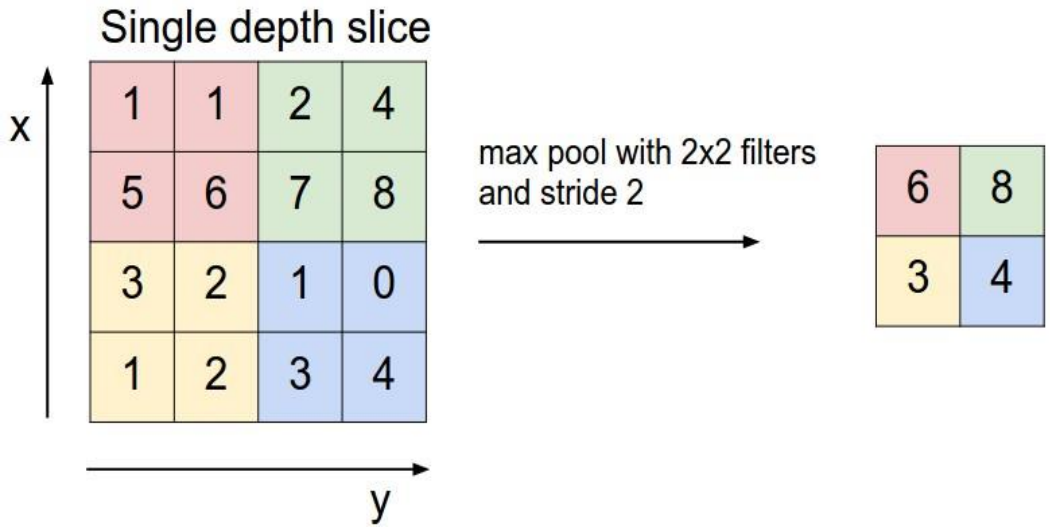


Figura 2.11 Operazione di MAX pooling con filtro 2x2 e passo 2 su un depth slice

Prendendo in esame la figura 2.11, notiamo come viene utilizzato un filtro di dimensione 2x2 con passo 2: viene sempre preso il valore massimo, considerato più importante ai fini dell'analisi, scartando quelli più bassi, diminuendo drasticamente la dimensione iniziale.

Vi è un secondo algoritmo che è quello dell'*Average Pooling*, ovvero a volte è conveniente, anziché estrarre il massimo dei valori considerati, prenderne il valor medio; questa tecnica è molto usata nelle recenti architetture in presenza dei *Fully Connected layer*.

Più in generale un Pooling Layer:

- Accetta un volume di dimensione $W_1 \times H_1 \times D_1$;
- Richiede due iperparametri:
  - L'estensione spaziale F;
  - Il passo S.
- Produce un volume di dimensione $W_2 \times H_2 \times D_2$;
  - $W_2 = \frac{W_1 - F}{S} + 1$;
  - $H_2 = \frac{H_1 - F}{S} + 1$;
  - $D_1 = D_2$;
- Non introduce ulteriori parametri poiché calcola una funzione fissa dell'input;

- Per i Pooling Layer, non è operazione comune usare lo zero-padding sull'input.

2.5.1.6 Fully Connected Layer

Nel *Fully Connected layer*, i neuroni sono tutti interconnessi alle funzioni di attivazione del layer precedente, come visto nelle classiche reti neurali. L'output finale di un *Fully Connected layer* sarà un vettore di dimensione 1 x 1 x K, dove K rappresenta il numero di neuroni di cui è composto i layer, contente le attivazioni calcolate. È facile notare come, avendo un input organizzato in tre dimensioni e ottenendo un singolo vettore in output, non vi sia più la possibilità di applicare un altro *Convolutional Layer* a seguire. Infatti, la sua funzione principale è quella di raggruppare tutte le informazioni ottenute fino a quel momento, esprimendole con un singolo numero utile per la classificazione finale.

Possono essere presenti più *Fully Connected layer*, di cui l'ultimo avrà il parametro K pari al numero delle classi presenti nel dataset su cui sta lavorando la rete. I valori finali verranno infine passati all'*output layer* che avrà il compito di effettuare la classificazione utilizzando un'apposita funzione probabilistica.

Capitolo 3

Object Detection nel Deep Learning

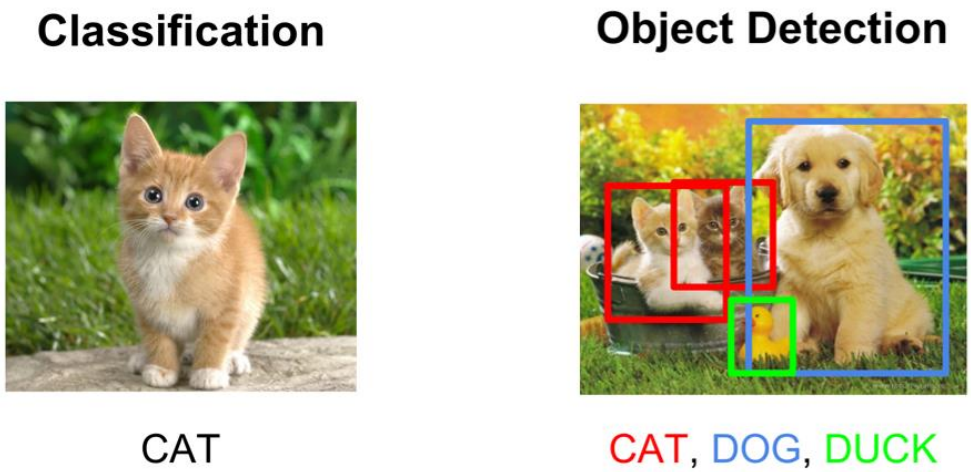


Figura 3.1 Differenza tra Image Classification e Object Detection. Fonte immagine e riferimenti [20].

Il lavoro di tesi si è focalizzato su un particolare campo del Deep Learning che è quello dell'*Object Detection.* Tale termine descrive, più in generale, una serie di problemi rivolti alla branca della *Computer Vision* in cui l'obiettivo è identificare gli oggetti presenti nelle immagini digitali disegnando dei box delimitatori attorno ad essi. Spesso tutto ciò viene confuso con il problema della classificazione: in generale, quando si vuole classificare un'immagine per una certa categoria, si parla di *Image Classification*; invece, quando si mira a identificare la posizione degli oggetti in un'immagine e contare, ad esempio, il numero di istanze di tale oggetto, si parla di *Object Detection*.

Spesso, tuttavia, i due scenari si sovrappongono, in quanto, se si vuole classificare un'immagine per una certa categoria, potrebbe accadere che l'oggetto, o i particolari che sono richiesti per una corretta classificazione,

sia troppo piccolo rispetto all'intera immagine. In tal caso si otterranno migliori performance con la *Object Detection* anche se non si era propriamente interessati su posizione o numero di istanze.

3.1 Stato dell'arte

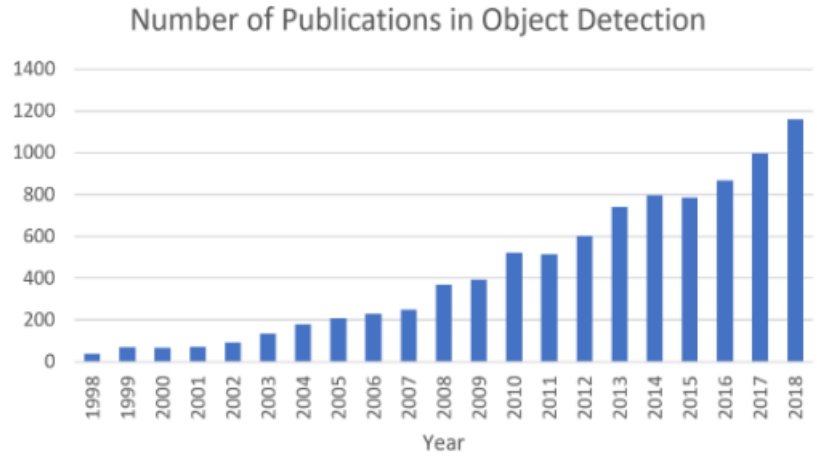


Figura 3.2 Numero di pubblicazioni sul tema dell'Object Detection dal 1998 al 2018.

Negli ultimi anni, il rapido sviluppo delle tecniche di Deep Learning ha portato di nuovo alla ribalta il campo dell'Object Detection, portandolo a notevoli passi in avanti [21]. Di fatto, l'Object Detection è ormai utilizzato in molteplici applicativi del mondo reale, ad esempio la videosorveglianza e la guida automatizzata.

3.2 Regioni di interesse

Di solito, si affrontano tre fasi nella costruzione di un framework per l'Object Detection:

1) Un modello, o algoritmo, è utilizzato per la generazione delle **Regioni di Interesse** (**RoI**) o **proposte di regione**. Tali proposte sono un gigantesco insieme di box identificatori (*bounding box*) che si espandono sull'intera immagine;

2) Le features visuali vengono estratte per ognuno dei bounding box e vengono valutate, al fine di determinare se e quali oggetti sono presenti nella proposta;
3) Infine, i box che si sovrappongono vengono combinati in un singolo box.

I passi appena esposti sono le tipiche fasi affrontate nello sviluppo di un *Object Detector* odierno; tuttavia, è una tecnica relativamente attuale, poiché le RoI vennero introdotte nel 2014 da R.Girshick.

3.3 Venti anni di ricerca

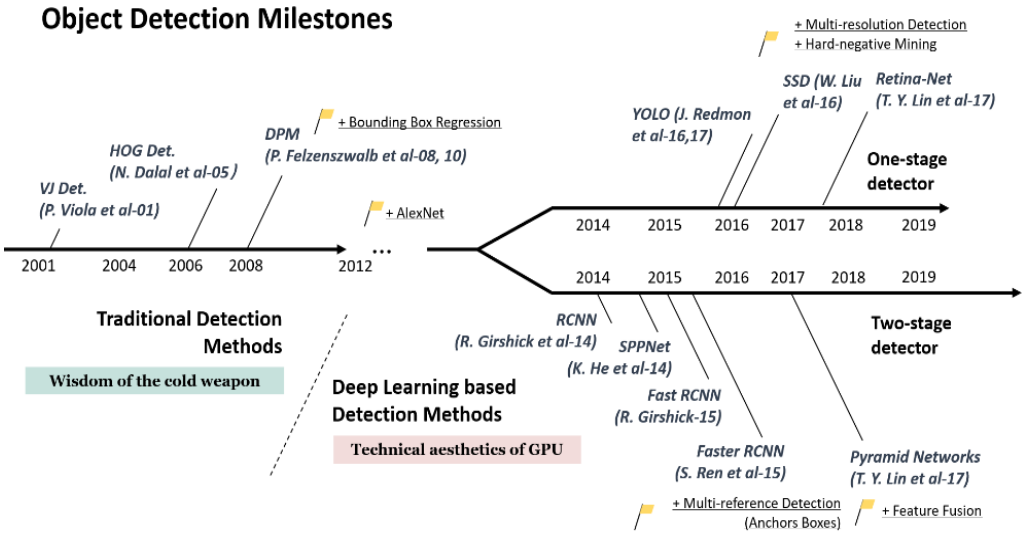


Figura 3.3 Pietre miliari dell'Object Detection dal 2001 al 2019.

Possiamo identificare, nelle ultime due decadi, due periodi attraversati dalla ricerca sull'Object Detection: "*Traditional Object Detector period*" (ante 2014) e "*Deep Learning based detection period*" (post 2014).

3.3.1 Viola Jones Detectors

Diciannove anni fa, P.Viola e M. Jones realizzarono il primo rilevatore capace di riconoscere, in tempo reale, dei volti umani, per la prima volta, senza vincoli, come il colore della pelle [22]. Venendo eseguito su una CPU Pentium III a 700MHz, il rilevatore era dieci o addirittura cento volte più veloce di ogni altro algoritmo di quel tempo in termini di accuratezza.

Seguendo una strategia "*straight-forward*", il rilevatore cercava tutte le possibili posizioni e dimensioni in un'immagine cercando se una qualsiasi finestra contenesse un viso. L'algoritmo ha migliorato drasticamente la sua velocità di rilevamento implementando tre tecniche:

1. *Integral Image*: l'immagine integrale è un metodo computazionale per velocizzare la selezione dei box o il processo di convoluzione. In particolare, è una struttura dati per il calcolo rapido della somma dei valori in un sottoinsieme rettangolare di una griglia. Per la rappresentazione delle feature venne utilizzata la *wavelet di Haar*;
2. *Feature Selection*: anziché utilizzare un insieme di filtri di Haar manualmente selezionati, gli autori *sfruttarono l'algoritmo di Adaboost* per selezionare un piccolo insieme di caratteristiche, considerate le più importanti per il rilevamento del viso;
3. *Detection cascades*: paradigma di rilevamento a molteplici stadi per ridurre il costo di computazione spendendo meno sulle finestre in background e più sui visi da individuare.

3.3.2 HOG detector

Histogram of Oriented Gradients (HOG), proposto nel 2005 da N, Dalal e B. Triggs [23], estrae, con buoni risultati, in termini di accuratezza e tempo di processamento, le feature dai colori dei pixel. L'applicazione dell'algoritmo può essere riassunta in cinque passi:

1. Pre-elaborare l'immagine: tutte le immagini devono essere identiche in termini di dimensione dei pixel in modo da ottenere risultati accurati e significativi;
2. Calcolare il gradiente nelle due direzioni principali con un filtro e, successivamente, usarlo per la computazione di magnitudine e direzione, poiché questi rimuove molto informazioni non essenziali;
3. L'immagine viene divisa in tante celle di dimensione 8x8 dove ogni magnitudine viene trasformata in un insieme di nove elementi privi di direzione e, nel caso in cui la magnitudine sia compresa tra due valori dell'insieme, questa è divisa proporzionalmente tra i due;
4. Una cella 2x2 trasla sull'immagine, dove ogni regione crea un vettore mono dimensionale di trentasei valori dai quattro istogrammi di

quattro celle e successivamente viene normalizzato ottenendo un valore di peso;

5. Il vettore composto dalla concatenazione di tutti questi vettori è la feature finale del HOG.

3.3.3 Deformable Part-based Model (DPM)

DPM, vincitore del VOC-07,-08 e -09, fu l'apice dei metodi tradizionali di Object Detection [24]. Proposto da P. Felzenszwalb nel 2008, fu un'estensione dell'HOG detector e, successivamente, molteplici miglioramenti vennero eseguiti da R. Girshick.

DPM segue il paradigma "dividi et impera", in cui l'addestramento può essere semplicemente considerato come l'apprendimento di un metodo appropriato per decomporre l'oggetto e la fase di inferenza è considerata come un insieme di rilevamenti su parti differenti dell'oggetto. Ad esempio, il problema di individuare un'auto può essere considerato come l'individuazione delle ruote, il suo corpo ecc.

In particolare, un tipico detector DPM è composto da un filtro "radice" (*root-filter*) e un numero definito di filtri secondari (*part-filter*), per i quali, anziché specificare manualmente le configurazioni, ad esempio dimensione e posizione, viene sviluppato un debole metodo di Apprendimento Supervisionato secondo il quale tutte le loro configurazioni possono essere apprese automaticamente come variabili latenti.

Nel 2010 Felzenswalb e Girshick furono premiati con il "lifetime achievement" dalla PASCAL VOC.

3.4 CNN e Object Detection

Con l'evoluzione delle reti neurali convoluzionali, nel 2014 R.Girshick *et al*. proposero le regioni con le caratteristiche delle CNN per l'object detection o, più semplicemente, le R-CNN [25].

3.4.1 Regional CNN

Si parte con l'estrazione di un insieme di possibili box identificatori, le "proposte", con l'uso del *Selective Search* [26]. Ogni proposta viene poi riscalata a una dimensione fissa e data in pasto al modello della CNN pre-addestrata per estrarne le feature. Infine, dei classificatori lineari SVM (Support Vector Machines) vengono usati per predire la presenza di un oggetto in ogni regione e riconoscerne la categoria.

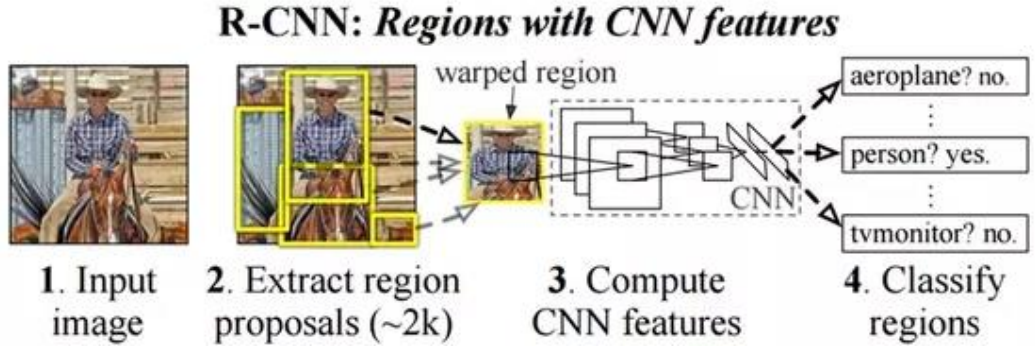


Figura 3.4 Funzionamento delle R-CNN.

Nonostante i progressi raggiunti, gli svantaggi delle R-CNN erano evidenti: il calcolo ridondante delle feature su di un gigantesco numero di proposte causava una lentezza inaccettabile.

3.4.2 SPPNet

Vennero così proposte, sempre nel 2014, le *Spatial Pyramid Pooling Networks* (*SPPNet*) da K. He *et al*. [27]. Le precedenti CNN richiedevano che l'immagine di input avesse dimensione fissa, ma, con l'introduzione dello *Spatial Pyramid Pooling layer*, le CNN erano capaci di generare una rappresentazione a dimensione fissa indipendentemente dalla dimensione dell'immagine, senza dover, dunque, riscalare quest'ultima. Con queste reti, le feature map venivano calcolate una sola volta per l'intera immagine e, successivamente, venivano generate rappresentazioni di dimensione fissa per regioni arbitrarie utilizzate per addestrare gli identificatori. Tuttavia, l'addestramento delle reti continuava ad essere multistadio e venivano affinati solo i layer Fully Connected, ignorando i precedenti.

3.4.3 Fast R-CNN

Nel 2015, R.Girshick propose un ulteriore miglioramento alle reti "region proposal" che permise di svolgere simultaneamente la fase di addestramento di un identificatore e di un box regressor utilizzando una configurazione della rete identica per entrambi [28].

Ghirshick realizzò che, per ogni immagine, una moltitudine di proposte di regioni si sovrapponeva inevitabilmente, causando lo svolgimento delle stesse operazioni di convoluzioni migliaia di volte. Fu così che introdusse la tecnica del *RoIPool* che prevede vengano forniti due input:

- Una feature map ottenuta dalla CNN;
- N proposte di RoI, ognuna composta da 5 valori: il primo indica l'indice e i restanti 4 sono le coordinate (tipicamente una prima coppia che indica l'angolo in alto a sinistra e una seconda che indica l'angolo in basso a destra).

La tecnica prende ogni RoI dall'input e una sezione della feature map che corrisponde a tale RoI e, successivamente, la converte in una sezione di dimensione fissa che dipende soltanto da alcuni parametri specificati: larghezza(W), altezza(H), scala spaziale.

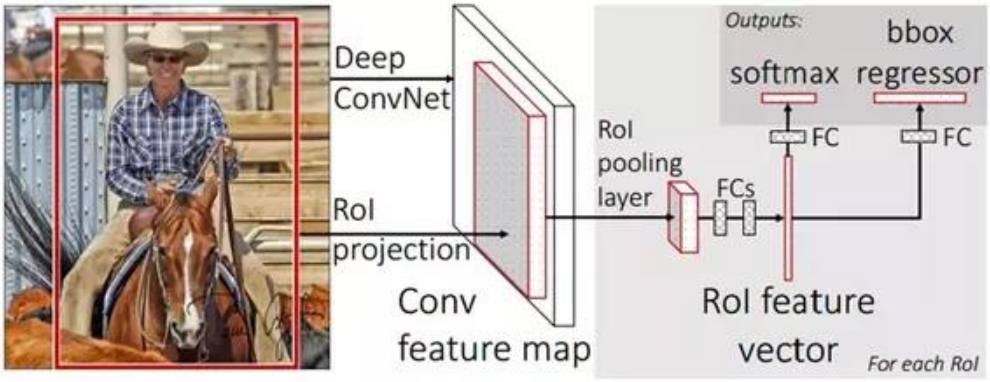


Figura 3.5 Flusso di lavoro della Fast R-CNN. Fonte immagine: [29].

Larghezza e altezza sono iperparametri decisi in base al problema in questione e indicano il numero di griglie in cui la proposta va divisa, mentre la scala spaziale ridimensiona la proposta in base alle dimensioni della feature map. Dunque, per ogni proposta, si prende la sezione di feature map,

la si divide in blocchi di dimensione W*H e si estrae il valore massimo di ogni blocco. L'output avrà dimensione fissa W*H per le N proposte.

Nonostante i miglioramenti in accuratezza e velocità, i colli di bottiglia e la velocità nell'identificazione delle proposte erano ancora un problema.

3.4.4 Faster R-CNN

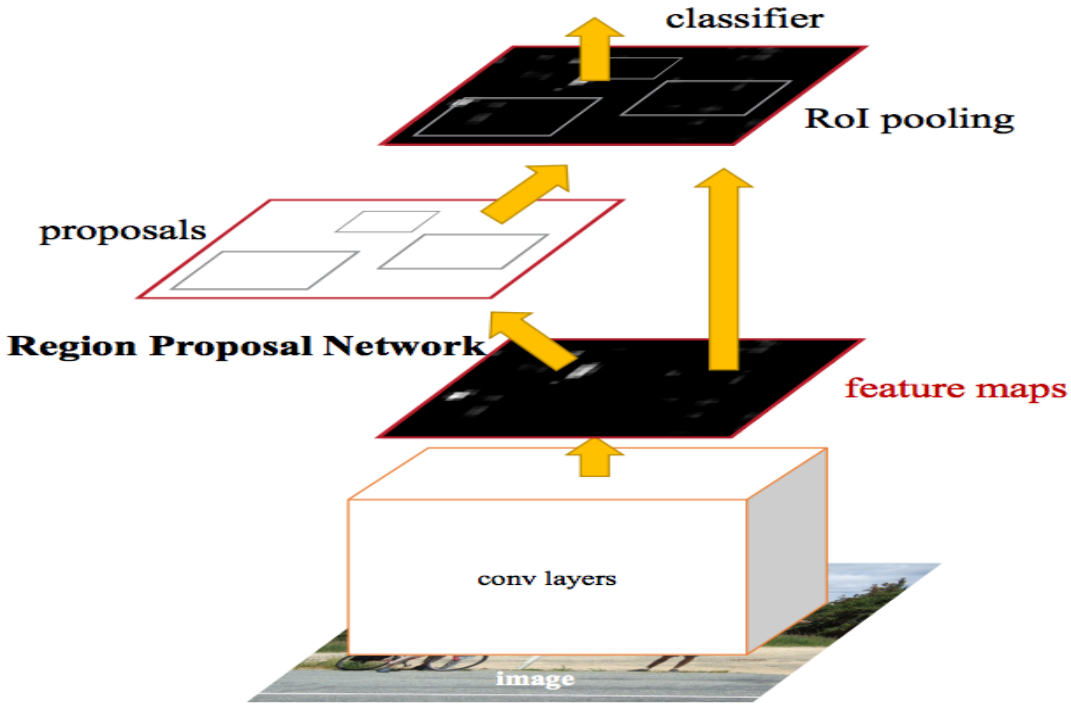


Figura 3.6 Architettura della Faster R-CNN.

Sempre nel 2015, poco dopo l'introduzione delle Fast RCNN, S. Ren. *et al*. proposero un nuovo tipo di identificatore, il primo capace di eseguire l'identificazione con un valore prossimo a quello real-time [30]. La novità, come possiamo vedere in figura 3.6, era l'introduzione della *Region Proposal Network* (*RPN*), una sorta di pipeline che si poggia sulla Fast R-CNN indicandole, però, cosa va osservato. Di fatto, la struttura della Fast R-CNN non cambia, ma viene aggiunta una rete Fully Connected che permette di processare un'immagine ogni 10ms. Nello specifico, la RPN lavora traslando una finestra sulla feature map della CNN e, per ogni finestra, genera k-potenziali box identificatori e quantifica l'accuratezza della presenza di un

oggetto all'interno di ognuno. Questi box verranno, successivamente, forniti al Fast R-CNN detector per eseguire la classificazione.

3.4.5 Mask R-CNN

Nel 2017, He K. *et al*. proposero un'ottimizzazione della Faster R-CNN chiamata *Mask R-CNN* [31]: tale variante utilizza sempre la procedura a due stadi della precedente, con il primo stadio, ovvero l'utilizzo della RPN, sostanzialmente identico; nel secondo stadio, invece, oltre a predire in parallelo la classe e l'offset del box, per ogni RoI, la rete genera una maschera binaria la quale specifica se un pixel è parte o no di un oggetto. Gli autori notarono che, eseguendo le modifiche sull'architettura originale della Faster R-CNN, le RoI selezionate era disallineate rispetto alla regione dell'immagine originale. Il problema venne risolto sviluppando il metodo del *RoIAlign*.

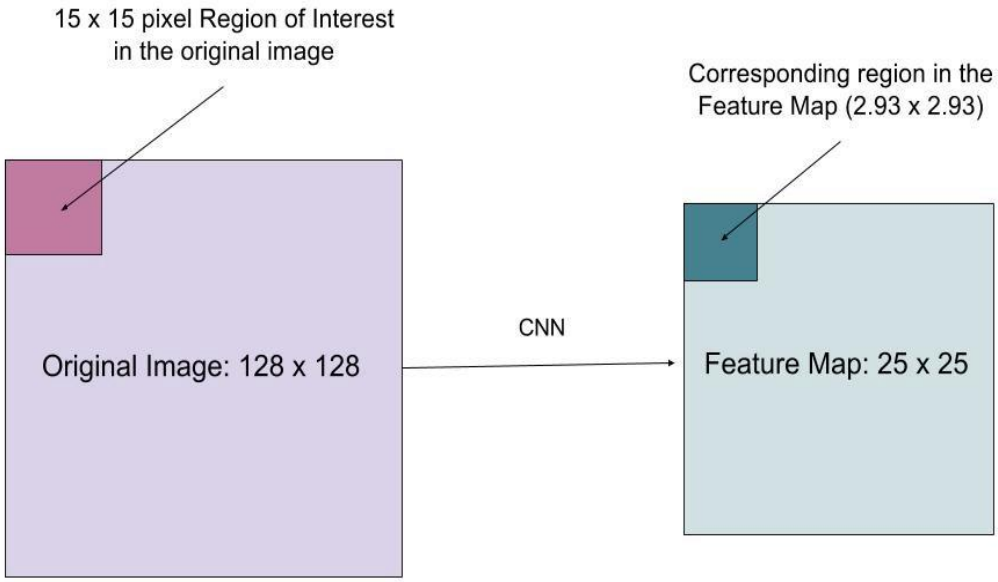


Figura 3.7 Esempio di applicazione di RoIAlign. Fonte immagine [32]

Osserviamo la figura 3.7: abbiamo un'immagine 128x128, una feature map di 25x25 e la nostra RoI che sarebbe la porzione 15x15 nell'angolo in alto a sinistra dell'immagine originale; sapendo che ogni pixel dell'immagine corrisponde a circa 25/128 pixel nella feature map, per selezionare 15 pixel nell'immagine originale selezioniamo 15 * 25 / 128 ~= 2,93 pixel.

Nel RoIPool, si sarebbe arrotondato il valore a 2 pixel causando il disallineamento, invece nel RoIAlign si usa l'interpolazione bilineare così da sapere precisamente cosa c'è al pixel 2,93.

Una volta generate le maschere, la Mask R-CNN le unisce alla classificazione e ai box identificatori della Faster R-CNN generando delle precise segmentazioni.

3.4.6 You Only Look Once

YOLO [33] venne proposta da R.Joseph *et al*. nel 2015 e si può capire subito dal suo nome come venga abbandonato lo schema di identificazione delle proposte e verifica di queste. Nella YOLO è prevista l'applicazione di una singola rete neurale che divide in regioni l'intera immagine e prevede i box delimitatori e le probabilità di ognuna simultaneamente. Combinando insieme i risultati, vengono estratti i box identificatori finali.

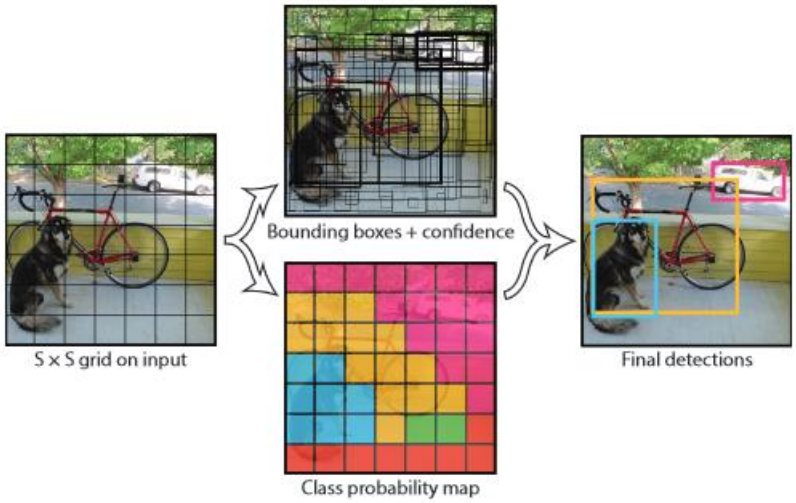


Figura 3.8 Flusso di lavoro di YOLO.

La caratteristica principale di YOLO è la sua incredibile velocità, basti pensare che con il dataset VOC07 la sua velocità era di 155 frame per secondo (fps). Tuttavia, YOLO soffre di scarsa accuratezza rispetto alle altre reti, in particolare con oggetti di piccole dimensioni. Dopo la prima versione, ne venne rilasciata una seconda migliorata nel 2017, *YOLOv2* [34], per poi seguirne una terza nel 2018, *YOLOv3* [35].

3.4.7 Single Shot Multibox Detector

SSD [36] venne proposto da W.Liu *et al*. nel 2015 introducendo le tecniche di individuazione multi-referenza e multi-risoluzione. SSD sta per:

- *Single Shot*: l'identificazione dell'oggetto e la relativa classificazione viene eseguita in un singolo passo in avanti della rete;
- *MultiBox*: nome della tecnica per la regressione dei box identificatori sviluppata da Szegedy *et al*.;
- *Detector*: la rete è un object detector volto a classificare gli oggetti rilevati.

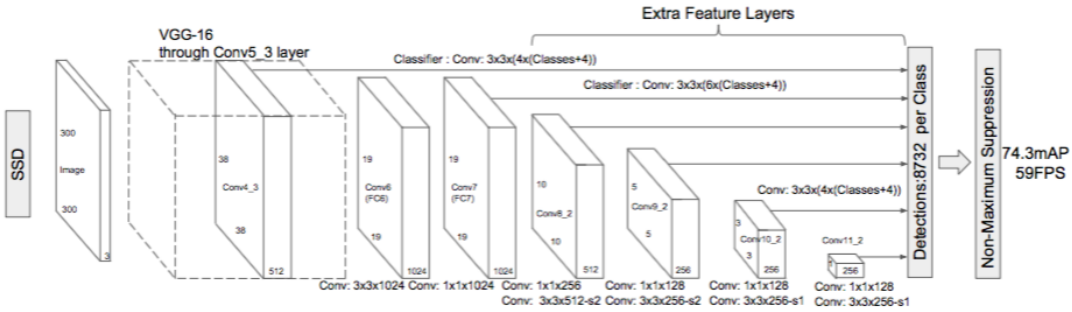


Figura 3.9 architettura di un detector SSD.

L'architettura si basa su quella della rete *VGG-16*, eliminando però i layer completamente connessi; le ragioni per la quale è stata utilizzata tale rete come base sono le sue performance nella classificazione di immagini ad alta qualità e la sua popolarità nei problemi in cui la tecnica del *Transfer Learning* aiuta a migliorare i risultati. Invece che i layer completamente connessi, è stato implementato un set di convolutional layer ausiliari così da estrarre le feature per scale multiple e diminuire progressivamente la dimensione dell'input per ogni layer seguente.

3.5 Metriche

Quando si valutano le prestazioni di una rete volta alla Object Detection, si usano delle precise metriche, molto differenti da quelle delle reti volte alla classificazione, in quanto le variabili non sono booleane, bensì pixel.

3.5.1 IoU

Si può considerare *Intersection over Union* (IoU) come la metrica di base per la valutazione di una rete volta all'Object Detection; IoU misura l'accuratezza del detector su di un particolare insieme test. Richiede solo due elementi per ogni immagine da valutare:

- Il *ground-thruth* box, ovvero il box identificatore generato dal programmatore stesso nel momento in cui viene generato il dataset e rappresenta la dimensione minima del box che racchiude l'oggetto;
- Il box predetto, ovvero l'ouput del modello.

Matematicamente, il parametro IoU equivale a:

$$IoU = \frac{Area\ d'intersezione}{Area\ dell'unione}$$

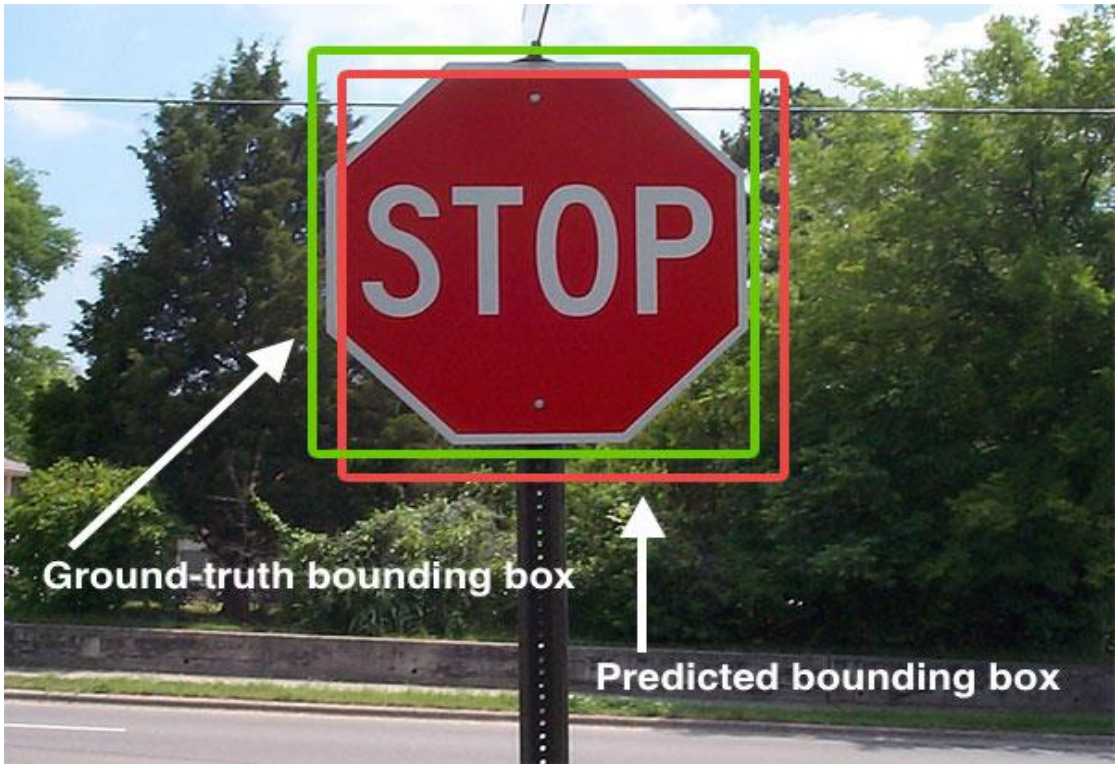


Figura 3.10 Esempio di IoU. Fonte immagine [37].

IoU si usa principalmente perché è impossibile che le coordinate (x,y) del box predetto siano identiche a quelle del ground-truth box per via della moltitudine di parametri che vengono considerati nella Object Detection. Per

tale motivo, si è definita una metrica che "premiasse" i box che si sovrappongano il più possibile con il ground-truth box.

3.5.2 Average Precision

Spesso, tra le metriche utilizzate nella valutazione di un detector, si troverà la sigla AP, che sta per *Average Precision*, ovvero la precisione media, ma bisogna prima introdurre alcuni concetti.

- Definiamo "*confidence*" la probabilità che un box contenga l'oggetto;
- Definiamo *True Positive* (TP) un'identificazione che soddisfi tre condizioni:
    - Il valore di confidence è maggiore di una certa soglia;
    - La classe predetta corrisponde a quella corretta;
    - Il valore di IoU è maggiore di una certa soglia.
- Definiamo *False Positive* (FP) un'identificazione che viola una delle ultime due condizioni;
- Definiamo *False Negative* (NF) un'identificazione che dovrebbe essere corretta ma che ha un valore di confidence inferiore alla soglia;
- Definiamo precisione non altro che il numero di TP diviso per la somma tra TP e FP:

$$precision = \frac{TP}{TP + FP}$$

- Definiamo *Recall* non altro che il numero di TP diviso per la somma tra TP e FN:

$$recall = \frac{TP}{TP + FN}$$

Impostando molteplici valori di soglia per la confidence, si ottengono differenti coppie di precisione e recall. Rappresentando le due metriche su di un diagramma cartesiano otteniamo la curva precision-recall.

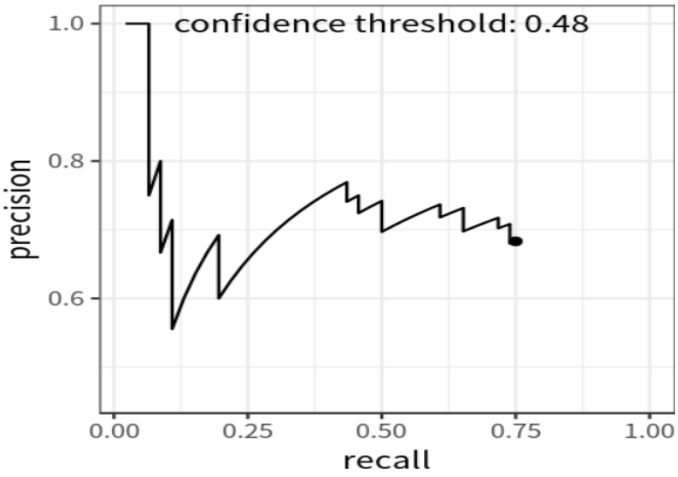


Figura 3.11 Esempio di curva precision-recall. Fonte immagine [38].

La precisione media, la quale è basata proprio su tale curva, è la precisione media tra tutti i livelli di recall. Per ridurre l'impatto dell'andamento ondulatorio sul valore di AP, la curva viene prima interpolata. La precisione interpolata ad un certo livello r è definita come la precisione più alta trovata per ogni livello $r' > r$:

$$p_{interp}(r) = \max p(r') \ \ con\, r' > r$$

Infine, la precisione media può essere definita come l'area sottostante la curva precision-recall interpolata che può essere calcolata con la seguente formula:

$$AP = \sum_{i=1}^{n-1} (r_{i+1} - r_i)\, p_{interp}(r_{i+1})$$

Dove $r_1, \ldots, r_n$ sono i livelli di recall dove la precisione è stata interpolata.

3.5.3 Mean Average Precision

La AP riguarda una singola classe; per quanto riguarda tutte le K classi presenti, definiamo la Mean Average Precion (mAP) come:

$$mAP = \frac{\sum_{i=1}^{K} AP_i}{K}$$

Capitolo 4

Intel® Neural Compute Stick 2

Figura 3.1 Intel® Neural Compute Stick 2.

Questo capitolo consiste in uno studio dell'*Intel® Neural Compute Stick 2*, sia lato software che hardware, così da capirne le caratteristiche e le potenzialità, mostrando un confronto con alcuni dispositivi analoghi e con la CPU Intel® Core™ i7-6700HQ 2.7Ghz. Vengono mostrati, inoltre, l'esecuzione di alcune applicazioni fornite dal toolkit *OpenVINO*™ e un esempio di applicazione scritta in linguaggio Python basata sul paradigma Client-Server con l'utilizzo dei socket, in cui viene messo in luce come si possa sviluppare un applicativo che, fornita una richiesta, esegua l'inferenza con la Intel® Neural Compute Stick 2 generando una risposta.

4.1 Il dispositivo

L'*Intel® Neural Compute Stick 2* (NCS2) è un dispositivo embedded dalle dimensioni e forme di una chiavetta USB ma dalle caratteristiche totalmente

diverse perché contiene un processore, una microarchitettura acceleratrice una memoria RAM e supporta molteplici software.

4.1.1 L'hardware

Per quanto riguarda l'hardware [39][40][41][42], le sue caratteristiche principali sono:

- Processore: Intel® Movidius Myriad X *Vision Processing Unit* (VPU);
- Connettività: USB 3.0 Type-A;
- Dimensione: 72.5 mm x 27 mm x 14 mm.

Più nel dettaglio, il sopracitato *Myriad X* VPU, il quale è costruito su *FinFET* [43] a 16nm, anziché a 28nm come il suo predecessore il Myriad 2, ha 16 core programmabili, quattro in più rispetto alla precedente NCS, basati sull'architettura 128-bit *Streaming Hybrid Architecture Vector Engine* o, più semplicemente *SHAVE*, la quale è una microarchitettura acceleratrice progettata da Movidius per i loro *vision processor* [44].

Inoltre, la NCS2 ha una memoria *on-chip* di 2,5MB, larghezza di banda per la memoria da 2,5 MB a 450 GB/s e due core RISC *general-purpose* assieme a nuovi acceleratori di visione a basso consumo che includono un blocco stereo capace di processare flussi duali a 720p fino a 180Hz e una *signal processor pipeline* integrata sincronizzabile con codifica *hardware-based* per una risoluzione video fino a 4K tra gli otto sensori (sei nel Myriad 2).

In particolare, Myriad X è la prima VPU a supportare il *Neural Compute Engine* di Intel®, un acceleratore hardware dedicato con supporto nativo alla precisione FP16 e punto fisso a 8-bit, grazie al quale si possono raggiungere mille miliardi di operazioni al secondo (TOPS) di picco nell'inferenza DNN, sullo sfondo delle prestazioni complessive teoriche di Myriad X di oltre quattro TOPS, performance dieci volte superiori al Myriad 2, il quale esegue "solo" da 1 a 1,5 TOPS.

Il Myriad X è disponibile in due versioni che forniscono differenti tipi di memoria: la MA2085 che non ha una memoria *on-package* ma espone

un’interfaccia per la memoria esterna, e la MA2485, versione utilizzata nello svolgimento del lavoro di ricerca per questa tesi, che supporta 4Gb di memoria LPDDR4 *in-package*, la quale è un grosso passo avanti rispetto alla LPDDR3 supportata dal precedente modello.

A livello di interfacce, il Myriad X supporta lo USB 3.1 e PCIe 3.0, entrambi nuovi nella famiglia delle VPU Myriad.
Tutto ciò è alimentato da meno di 2W di potenza, come nel Myriad 2, più precisamente entro 1 W.

Movidius Myriad Family VPUs		
	Myriad 2	Myriad X
Compute Capacity	>1 TOPS	>4 TOPS
Vector Processors	12x SHAVE Processors	16x SHAVE Processors
CPUs	2x LEON4 cores (RISC; SPARC V8)	2x LEON4 cores (RISC; SPARC V8)
On-chip Accelerators	~20 image/vision processing accelerators	20+ image/vision processing accelerators Neural Compute Engine (DNN accelerator)
Neural Network Capability	1st Gen DNN Support (Up to 100 GFLOPS)	Neural Compute Engine (Up to 1 TOPS)
On-chip Memory and Bandwidth	2 MB (400GB/sec)	2.5 MB (450GB/sec)
DRAM Support	Max: 8Gb LPDDR2 (533MHz, 32-bit) LPDDR3 (933MHz, 32-bit)	Max: 16Gb LPDDR4 (1600MHz, 32-bit)
DRAM Configurations	1Gbit LPDDR2 (MA215X) 4Gbit LPDDR3 (MA245X)	No in-package memory (MA2085) 4Gbit LPDDR4 (MA2485)
Encoder/Codec	VGA, 720p, 1080p, H.264 (software encoder)	M/JPEG 4K at 60Hz encoder H.264/H.265 4K at 30Hz encoder
Key Interfaces	12x MIPI lanes (DPHY 1.1) USB 3 SPI I2S SD 1GbE	16x MIPI lanes (PHY 1.2) USB 3.1 Quad SPI I2S 2x SD 10GbE PCIe 3.0
Process	28nm HPC+/HPC/HPM (TSMC)	16nm FFC (TSMC)
Package	6.5mm x 6.5 mm (MA215X) 8mm x 9.5 mm (MA245X)	8.1mm x 8.8mm (MA2085, MA2485)

Figura 4.2 Schema riassuntivo delle differenze tra Myriad 2 e Myriad X. Fonte immagine: [40].

4.1.2 Il software

Per quanto riguarda il lato software [45], Intel® fornisce:

- Distribuzione Intel® del toolkit OpenVINO™;
- Supporto dei seguenti sistemi operativi:
  - Ubuntu 16.04.3 LTS (64 bit);
  - CentOS 7.4 (64 bit);
  - Windows 10 (64 bit);
  - macOS 10.14.4 (o seguenti);
  - Raspbian (solo target);
  - Altri (via OpenVINO toolkit).

La NCS2 supporta i seguenti framework per lo sviluppo delle reti neurali:

- TensorFlow;
- Caffe;
- Apache MXNet;
- Open Neural Network Exchange (ONNX);
- PyTorch (via ONNX);
- PaddlePaddle (via ONNX);

4.2 Il toolkit OpenVINO™

OpenVINO™ (*Open Visual Inference and Neural network Optimization*) è un toolkit gratuito per l'ottimizzazione di modelli di Deep Learning da un framework e la loro distribuzione, usando l'*Inference Engine* su hardware Intel®. Le informazioni che seguono sono tratte dall'esperienza avuta con il toolkit durante lo sviluppo del lavoro di tesi unita alla documentazione ufficiale [46].

Innanzitutto, OpenVINO™ ha i seguenti requisiti:

- Hardware
  - Dalla sesta alla decima generazione di processori Intel® Core™;
  - Famiglia Intel® Xeon® v5;
  - Famiglia Intel® Xeon® v6;

    - Intel® Movidius™ Neural Compute Stick
    - Intel® Neural Compute Stick 2
    - Intel® Vision Accelerator Design con le VPU Intel® Movidius™.
- Software
    - Microsoft Visual Studio con C++ 2019, 2017 o 2015 con MSBuild;
    - CMake dalla versione 3.4 in su a 64 bit;
    - Python 3.6.5 a 64 bit.

Per il caricamento del modello di una rete neurale sulla NCS2, *OpenVINO™* fornisce il *Model Optimizer*, un tool da linea di comando *cross-platform* che facilita la transizione dall'ambiente di training a quello di deployment, esegue un'analisi statica del modello e adegua i modelli di deep learning per una corretta esecuzione sul dispositivo scelto.

Il processo del Model Optimizer presuppone che si abbia un modello di rete neurale già addestrato che utilizzi uno dei framework supportati; il tool è capace, inoltre, di rilevare quale framework supporta il modello e, in caso non siano installati i rispettivi moduli sulla propria macchina, suggerirà di usare degli script *batch* (*.bat*), chiamati

"*install_prerequisites_<framework>*"

forniti anch'essi con il toolkit *OpenVINO™*, per colmare le lacune e procedere con la conversione.

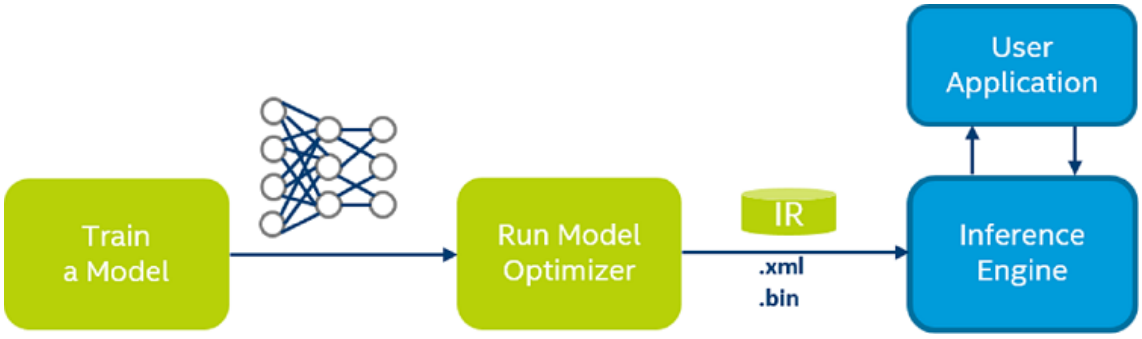


Figura 4.3 Diagramma riassuntivo per la conversione del modello ed esecuzione con l'Inference Engine.

Il Model Optimizer produce la *Intermediate Representation* (IR) della rete, la quale può essere letta, caricata e fatta inferenza con *l'Inference Engine*. La IR è composta da due file con i seguenti formati, rispettivamente:

- *.xml*: descrive la topologia della rete;
- *.bin*: contiene i pesi e i bias dei dati binari.

Con OpenVINO™ viene fornita, inoltre, la *Inference Engine API*, una API unificata che permette alte prestazioni d'inferenza su dispositivi Intel®, quali CPU, processori grafici e modelli di NCS. L'*Inference Engine* è una libreria C++ (esiste anche la controparte Python) con un set di classi C++ per eseguire l'inferenza sui dati di input per ottenere i risultati e fornisce una API per leggere correttamente la IR, impostare i formati input e output ed eseguire il modello sui dispositivi.

Oltre a ciò, OpenVINO™ fornisce agli sviluppatori, tra i vari tool, il *Model Downloader,* con altrettanti script, basati su file di configurazione, che automatizzano alcuni compiti del programmatore, tra cui:

- Il *Model Converter* (*converter.py*): converte i modelli che non sono nel formato IR in tale formato utilizzando il Model Optimizer;
- Il *Model Information Dumper* (*info_dumper.py*): stampa le informazioni riguardo i modelli in un formato stabile machine-readable.

Il *Model Downloader*, in particolare, è uno script Python che si trova nella cartella *<INSTALL_DIR>\deployment_tools\tools\model_downloader* e prende il nome di *downloader.py* il quale permette di scaricare da sorgenti web i file dei modelli di alcune reti neurali pre-addestrate e, se necessario, correggerli in modo tale che siano utilizzabili dal Model Optimizer.

Per utlizzare il Model Downloader è richiesto:

- Python (dalla versione 3.5.2)
- L'installazione delle dipendenze con il comando:
  ```
  python3 -m pip install --user -r ./requirements.in
  ```

Infine, lanciando lo script *downloader.py* seguito dall'opzione "-- name" e il nome del modello che si vuole scaricare, verrà avviato il download di due file distinti aventi i seguenti formati:

- *.prototxt*: contiene la topologia della rete;
- *.<framework_name>model*: è il modello, con pesi e bias, della rete pre-addestrato e compilato nel framework disponibile.

Successivamente, per eseguire il caricamento del modello sul dispositivo, si utilizza il Model Converter per convertire la rete nel formato IR lanciando il *converter.py*, specificando il *path* dove si trovano i file del modello, oppure direttamente il Model Optimizer usando lo script generico *mo.py* specificando, come argomento, che la precisione deve essere **FP16**, nel caso in cui il dispositivo il dispositivo finale su cui caricare il modello sia la NCS2, altrimenti **FP32**, nel caso della CPU.

Ad esempio, per convertire il modello della *GoogleNetV1* scritto con il framework *Caffe* da caricare su NCS2, il comando da lanciare è il seguente:

```
mo.py --data_type FP16 --input_model googlenet-v1.caffemodel --input_proto googlenet-v1.prototxt
```

A questo punto si ha quanto necessario per poter sviluppare un'applicazione che sfrutti le Inference Engine API per eseguire l'inferenza sulla NCS2 con il modello appena convertito.

Oltre a quanto riportato, nell'installazione di OpenVINO™ sono inclusi:

- *OpenCV*: *Open Source Computer Vision Library* è una libreria software multipiattaforma per la Computer Vision, sviluppata da Intel®;
- *OpenCL™*: *Open Community Language* è un framework per la scrittura di programmi e la loro esecuzione tra piattaforme eterogenee.

OpenVINO™ fornisce, inoltre, una serie di samples e demo per l'utilizzo dei propri modelli di rete neurale con l'Inference Engine scritti sia in linguaggio C++ che Python. Per i progetti C++, vi è un file in formato "*.batch*", fornito appositamente per la compilazione e creazione dei relativi eseguibili, che si trova seguendo il path:

<INSTALL_DIR>\deployment_tools\inference_engine\samples\build_samples_msvc.bat

4.3 Confronto con dispositivi simili

Grazie ad A.Alasdair [47] possiamo avere un'idea chiara di quali siano le prestazioni della NCS2 rispetto al suo predecessore e ad altri analoghi dispositivi. Il confronto è stato eseguito tra la *Coral Dev Board*, la *NVIDIA Jetson Nano*, il *Coral USB Accelerator* con un *Raspberry Pi*, l'originale *Intel® Movidius Neural Compute Stick* con un Raspberry Pi, la seconda generazione *NCS2* sempre con un Raspberry Pi e, infine, un *Apple MacBook Pro (2016)* con Intel® Core™ i7 2.9 GHz quad-core e su un *Raspberry Pi 3*, *Model B+* senza accelerazione utilizzando, come reti neurali, la *SSD MobileNetV1* e la *SSD MobileNetV2*.

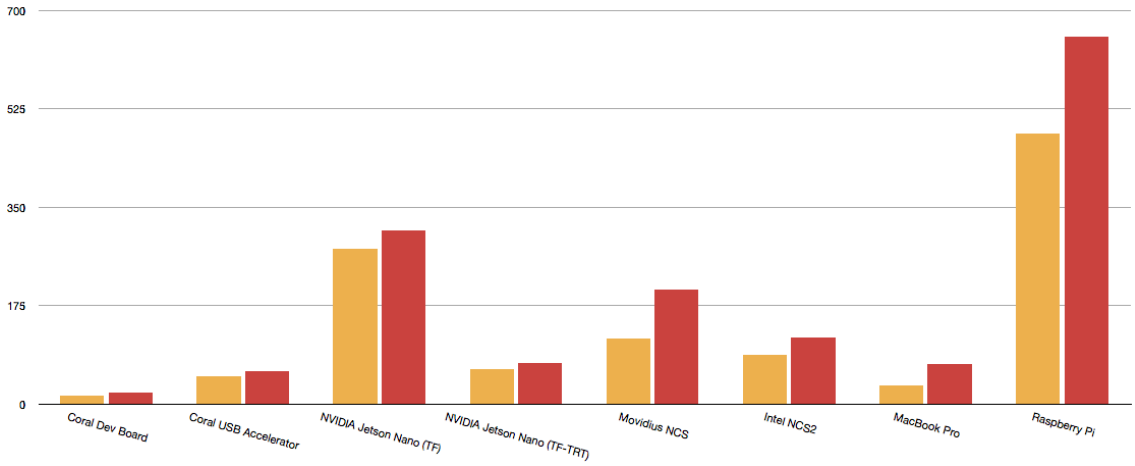


Figura 4.4 Sull'asse delle ordinate si trova la velocità d'inferenza in termini di millisecondi per la SSD MobileNetV1(arancione) e la SSD MobileNetV2 (rossa).

Entrambi i modelli sono stati allenati sul COCO dataset ed è stata usata un'immagine test da 3888x2916 pixel contenente due oggetti: una mela e una banana. L'immagine è stata scalata a 300x300 pixel prima di essere presentata ai modelli, di cui ognuno è stato eseguito 10.000 volte prima che un tempo d'inferenza medio venisse stimato.

Board	MobileNet v1 (ms)	MobileNet v2 (ms)	Idle Current (mA)	Peak Current (mA)	Price (US$)
Coral Dev Board	15.7	20.9	600	960	$149.00
Coral USB Accelerator	49.3	58.1	470	880	$74.99+$35.00
NVIDIA Jetson Nano (TF)	276.0	309.3	450	1220	$99.00
NVIDIA Jetson Nano (TF-TRT)	61.6	72.3			
Movidius NCS	115.7	204.5	500	860	$79.00+$35.00
Intel NCS2	87.2	118.6	480	910	$79.00+$35.00
MacBook Pro[1]	33.0	71.0	1570	1950	>$3,000
Raspberry Pi	480.3	654.0	410	1050	$35.00

[1] The MacBook Pro takes a +20V supply, all other platforms take a +5V supply.

Tabella 4.1 Confronto dei risultati; le ultime tre colonne indicano rispettivamente il consumo di corrente prima e durante l'inferenza e il prezzo dei dispositivi in dollari americani.

Il benchmark è stato eseguito usando modelli in *TensorFlow* o, per le piattaforme accelerate che non supportano TensorFlow come framework nativo, convertiti nell'appropriato framework nativo: per la Coral EdgeTPU-based hardware è stato usato *TensorFlow Lite*, per la Intel® NCS e la NCS2 il toolkit *OpenVINO™* e per le NVIDIA Jetson Nano sono stati utilizzati sia '*vanilla' Tensorflow* (con il supporto della GPU) e ancora TensorFlow ottimizzato usando il framework *NVIDIA TensorRT*. Inoltre, è bene sottolineare che la porta USB del Raspberry Pi 3 è una USB 2.0, non una USB 3.0; pertanto, alcuni risultati non coincidono con quelli dichiarati dai produttori e, purtuttavia, si può sottolineare come la velocità sia quasi dieci volte maggiore del Raspberry senza acceleratori.

Analizzando la tabella 4.1, notiamo come la NCS2 sia quasi il doppio più veloce del suo predecessore nell'eseguire l'inferenza, come nel caso della SSD MobileNetV2, e come permetta di eseguire un'inferenza cinque volte più veloce al Raspberry Pi. Da sottolineare, inoltre che la Coral Deb Board è pensata come dispositivo di valutazione per i *System-on-Module* (*SOM*) mentre il Coral USB Accelerator è più indicato per i data scientist, contesti applicativi molto differenti da quelli della NCS2. Diamo un'occhiata ora ai consumi elettrici:

Board	Idle Current (mA)	Peak Current (mA)
Coral Dev Board	600	960
Coral USB Accelerator	470	880
NVIDIA Jetson Nano	450	1220
Movidius NCS	500	860
Intel NCS2	480	910
MacBook Pro	1570	1950
Raspberry Pi	410	1050

Tabella 4.2 Misure di corrente, sia in stato inattivo che attivo.

Osservando la tabella 4.2, possiamo subito notare che, come da specifiche, la NCS2 non supera il valore di 1A d'intensità di corrente, a differenza del Raspberry Pi senza acceleratore e del MacBook Pro, ricordando, tuttavia, che quest'ultimo richiede 20V di alimentazione. Si può tranquillamente affermare, dunque, che l'utilizzo della NCS2 con il Raspberry Pi permette un risparmio energetico, anche se minimo, oltre che a nette migliori prestazioni d'inferenza.

Board	Peak External Temp (°C)	Peak CPU Temp (°C)
Coral Dev Board	52	64
Coral USB Accelerator	35	60
NVIDIA Jetson Nano	56	53
Movidius NCS	45	51
Intel NCS2	45	52
MacBook Pro	46	58
Raspberry Pi	58	74

Tabella 4.3 Picchi di temperatura misurata esternamente e per la CPU.

La tabella 4.3 riporta il picco delle temperature raggiunte dai dispositivi, rilevate sia esternamente ad essi che all'interno della CPU: si può subito notare come i dispositivi Intel® abbiano il valore minimo; inoltre, questi garantiscono al Raspberry Pi un valore di temperatura della CPU di ventitré gradi in meno.

4.4 Confronto con CPU

Uno dei samples forniti con il toolkit *OpenVINO™* è l'applicazione *benchmark_app* che esegue una stima delle performance d'inferenza di un modello di rete neurale sui dispositivi supportati dal plugin, eseguendo quattro operazioni d'inferenza, in modo asincrono o sincrono, in base ai parametri specificati in fase di lancio, su quattro immagini nell'arco di sessanta secondi, misurando il *throughput* in frame per secondi (fps), la latenza in millisecondi (ms) e il numero di iterazioni eseguite; nel caso in cui venga fornito un numero inferiore di immagini, l'applicazione creerà delle copie di quelle messe a disposizione raggiungendo il numero desiderato. Si è provveduto, dunque, a testare le prestazioni di tre modelli di reti neurali pre-addestrati scegliendoli arbitrariamente:

- Per la Image Classification è stata testata la *GoogleNetV1* e la *ResNet50*;
- Per l'Object Detection è stata testata la *SSD MobileNetV1*.

Le reti in questione sono state ottenute grazie al Model Downloader di OpenVINO™ e convertite nel formato IR con il Model Optimizer.

Il dispositivo con cui è stato effettuato il confronto è una CPU Intel® Core™ i7-6700HQ 2.6GHz.

A seguire, vi sono i riferimenti ipertestuali per le immagini utilizzate:

(1) https://github.com/movidius/ncappzoo/blob/master/data/images/nps_acoustic_guitar.png?raw=true
(2) https://upload.wikimedia.org/wikipedia/commons/b/b6/Felis_catus-cat_on_snow.jpg

(3) https://github.com/movidius/ncappzoo/blob/master/data/images/pic_057.jpg
(4) https://raw.githubusercontent.com/chuanqi305/MobileNet-SSD/master/images/000067.jpg

Inizialmente, si è eseguito l'applicazione solo con l'immagine (2) per analizzare il comportamento della CPU e della NCS2; di seguito possiamo osservare i risultati ottenuti con la ResNet50.

RESNET50	#Iterazioni	Latenza (ms)	Throughput (fps)
CPU	2616	91,2937	45,5328
NCS2	2124	113,134	35,3067

Tabella 4.4 Risultati di "benchmark app" con ResNet50 e immagine (2).

Con la ResNet50 si notano già le prime limitazioni della NCS2 rispetto alla CPU, in quanto la NCS2, con un numero di iterazioni minore, ha tempo di latenza più lungo e un throughput minore.

Vediamo di seguito, invece, la GoogleNetV1.

GOOGLENET V1	#Iterazioni	Latenza (ms)	Throughput (fps)
CPU	4600	52,3393	76,5491
NCS2	5072	47,3113	84,4498

Tabella 4.5 Risultati di "benchmark app" con GoogleNet v1 e immagine (2).

Con la GoogleNetV1, invece, possiamo osservare come la NCS2 riesce a tenere testa alla CPU, addirittura fornendo migliori risultati per numero di iterazioni eseguite, latenza e throughput.

SSD MOBILENET V1	#Iterazioni	Latenza (ms)	Throughput (fps)
CPU	6976	33,1164	116,203
NCS2	3436	69,8438	57,2002

Tabella 4.6 Risultati "benchmark app con SSD MobileNet v1 e immagine (2).

Possiamo notare, infine, come la CPU riesca ad essere molto più performante della NCS2 con una rete pre-trained volta alla Object Detection; tuttavia, questa rete è più indicata per essere veloce su dispositivi dalla bassa capacità computazionale; pertanto, è normale che la CPU dia risultati nettamente migliori.

Proviamo ora ad eseguire l'applicazione fornendo le quattro immagini, come da specifiche dell'applicazione.

RESNET50	#Iterazioni	Latenza (ms)	Throughput (fps)
CPU	2592	92,7857	43,0821
NCS2	2124	113,212	34,3166

Tabella 4.7 Risultati "benchmark app con ResNet50 e quattro immagini.

Con la ResNet50 si hanno risultati praticamente identici al caso precedente; il throughtput assume un valore leggermente inferiore.

GOOGLENET V1	#Iterazioni	Latenza (ms)	Throughput (fps)
CPU	5720	41,6226	95,2134
NCS2	5328	45,0364	88,7073

Tabella 4.8 Risultati "benchmark app con GoogleNet v1 e quattro immagini.

In questo caso possiamo notare che, con la GoogleNetV1, le performance della NCS2 non si discostano parecchio da quelle della CPU; tuttavia, la CPU ha registrato valori migliori della NCS2 a differenza del caso precedente.

SSD MOBILENET V1	#Iterazioni	Latenza (ms)	Throughput (fps)
CPU	7372	32,5409	112,8
NCS2	3432	69,9386	57,1068

Tabella 4.9 Risultati "benchmark app" con SSD MobileNet v1 e quattro immagini.

Anche in questo caso, i risultati non si discostano molto dal caso precedente, la SSD MobileNetV1 ottiene degli ottimi risultati eseguendo l'inferenza sfruttando la CPU, rispetto all'utilizzo della NCS2.

Tutti i test sono stati effettuati prendendo il risultato migliore sotto il punto di vista della latenza e del throughput per un massimo di dieci tentativi e stanno a dimostrare come, nonostante la NCS2 sia un valido dispositivo per eseguire l'inferenza, può raggiungere in alcuni casi le performance di una CPU di livello medio ma non potrà mai eguagliarle.

4.5 Applicazione Client-Server

In questo paragrafo, si mostra un esempio di applicazione Client-Server, scritta in linguaggio Python, in cui un mittente invia una richiesta di classificazione di un'immagine al server destinatario e questi, con il supporto della NCS2, esegue l'inferenza con i modelli a disposizione e restituisce il risultato al mittente. Il server utilizza le reti ResNet50 e GoogleNetV1 per eseguire la classificazione; dunque, è necessario avere i modelli già salvati nel formato IR e che il toolkit sia stato propriamente inizializzato.

Il protocollo si articola nel seguente modo:

1. Il client specifica il percorso all'immagine che ha salvata sul disco;
2. Il client tenta una connessione al server tramite i socket;

3. Stabilita la connessione, il cliente invia la dimensione del file preceduto dalla parola chiave “SIZE”
4. Il server controlla che il messaggio cominci con la parola “SIZE” e, in caso positivo, risponde al client con la parola chiave “SIZE_REC”, altrimenti notifica l’errore con la parola chiave “ERR” e chiude la connessione al client;
5. Il client, ricevuta la parola chiave “SIZE_REC” invia al server l’immagine in pezzi di 1024 bytes;
6. Il server non appena ha ricevuto l’intera immagine, ne confronta la dimensione con quella inviatagli in precedenza per controllare che non vi siano stati errori di trasmissione e, in caso di successo, risponde al client con il risultato della classificazione preceduto dalla parola chiave “+RES\r\n”;
7. Il client, infine, controlla che la risposta del server cominci con la parola chiave “+RES\r\n”, stampa a video il risultato inviatogli, in caso positivo, e, infine, invia al server la parola chiave “BYE” e chiude la connessione;
8. Il server controlla che se parola chiave ricevuta sia “BYE”, e in caso positivo, chiude la connessione con quel client.

Vengono riportati, di seguito, i punti salienti del codice per il client:

```
image = input("Please, specify path to image to send: ")

# check if file exists and if it’s a file
if os.path.isfile(image):
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
s:
        print('Trying connection to server...')
        s.connect((HOST, PORT))

        print('Connection estabilshed, sending request')
```

Figura 4.5 Connessione al server.

```
try:
      # open image
      # try-except block avoid the race condition
      with open(image, 'rb') as file:
         bytes = file.read()
         size = len(bytes)
         size_str = "SIZE %s" % size
         byt = size_str.encode()
         # send image size to server
         s.sendall(byt)
         answer = s.recv(1024)
         print('answer = %s' % answer.decode())
         # send image to server
          if answer.decode() == 'SIZE_REC':
             # looping and sending in chunks of 1024
             file.seek(0)
             bytes = file.read(1024)
             while bytes:
              s.send(bytes)

               bytes = file.read(1024)
```

Figura 4.6 Invio dell'immagine.

```
       # check server final answer
        answer = s.recv(1024)
        if answer.decode().startswith("+RES\r\n"):
           # server has received the full image
           print('Image successfully sent to server')
           answer = answer.decode()
           vAnswer = answer.splitlines()
           print(YELLOW + '\n **********' + NOCOLOR + '  Results
' +     YELLOW + '***********' + NOCOLOR)
           print(vAnswer[1])
           s.sendall(BYE_STRING.encode())
           s.close()
         elif answer.decode() == 'ERR':
            # server had a problem
             print("RECEIVED -ERR, CLOSING...")
             quit()
         else:
         # server had a problem
         print("RECEIVED -ERR, CLOSING...")

         quit()
```

Figura 4.7 Controllo del risultato e gestione dell'errore di comunicazione.

Per quanto riguarda il server:

```
# this function setup network parameters using the
Inference Engine APIs
def setup_network(ir, ie):

    # Select IRs to be used
    net = IENetwork(model=ir, weights=ir[:-3] + 'bin')
    # Set up the input and output blobs
    input_blob = next(iter(net.inputs))
    output_blob = next(iter(net.outputs))

    # Load the network and get the network shape
information
    input_shape = net.inputs[input_blob].shape
    output_shape = net.outputs[output_blob].shape

    exec_net = ie.load_network(network = net, device_name =
DEVICE)
    del net

    return input_shape, input_blob, output_blob, exec_net
```

Figura 4.8 Seup della rete con le IE API.

```
def preprocess_image(img, input_shape, mean):
    n, c, h, w = input_shape
    # Read in the image.
    print('Reading image %s' % img)
    img = cv2.cvtColor(numpy.array(img), cv2.COLOR_RGB2BGR)
    # img = cv2.imread(img)

    if img is None:
        print(RED + "\nUnable to read the image file." + NOCOLOR)
        return

    # Image preprocessing
    img = cv2.resize(img, (h, w))
    img = img.astype(numpy.float32)

    # Load the mean file and subtract the mean then tranpose the
image (hwc to chw)
    if mean is not None:
        ilsvrc_mean = numpy.load(mean).mean(1).mean(1)
        img[:, :, 0] = (img[:, :, 0] - ilsvrc_mean[0])
        img[:, :, 1] = (img[:, :, 1] - ilsvrc_mean[1])
        img[:, :, 2] = (img[:, :, 2] - ilsvrc_mean[2])
    transposed_img = numpy.transpose(img, (2, 0, 1))
    reshaped_img = transposed_img.reshape((n, c, h, w))

    return reshaped_img
```

Figura 4.9 Pre-processazione dell'immagine nelle dimensioni richieste dalla rete

```
def perform_inference(img, exec_net, input_blob, output_blob,
top):
    # Prepare Categories for age and gender networks
    with open(LABELS) as labels_file:
        label_list = labels_file.read().splitlines()
    res = exec_net.infer({input_blob: img})
    top_ind = numpy.argsort(res[output_blob], axis=1)[0, -
top:][::-1]
    result = ''
    print(YELLOW + '\n **********' + NOCOLOR + '  Results  ' +
YELLOW + '***********' + NOCOLOR)

    for k, i in enumerate(top_ind):
        result = result + (
                ' Prediction is ' + "%3.1f%%" % (100 *
res[output_blob][0, i]) + " " + label_list[int(i)] + "\n")
    print('')
    print( result )

    return result
```

Figura 4.10 Fase di inferenza.

```
## SETUP NETWORK AND LOAD TO NEURAL COMPUTE STICK 2 ##
    print("[INFO] : Loading googlenet-v1")
    input_shape, input_blob, output_blob, exec_net =
setup_network(googlenet_v1, ie)
    print("[INFO] : Loading resnet-50")
    input_shape2, input_blob2, output_blob2, exec_net2 =
setup_network(resnet_50, ie)
    print('[INFO] : Networks loaded, starting server...')

…

      try:
            data=sock.recv(BUFF_SIZE)
 # check if data expected is the image or a protocol message

              if not bool_str:
                  txt = str(data.decode())
               # check if client sent file size
                  if txt.startswith('SIZE'):
                    size = size_rec(txt)

     message_queues[sock].put(SIZE_STRING.encode())

                    if sock not in outputs:
                        outputs.append(sock)

                        bool_str = True
                       # check if client closed the connection
                    elif txt.startswith('BYE'):
                         print('Client closed the connection')
                         sock.close()
                         inputs.remove(sock)

                         break
```

Figura 4.11 Caricamento delle reti sulla chiavetta e controllo della ricezione delle parole chiave.

```
# A readable client socket has data
   elif data:
    print('Start writing file')
    # Building the final path of the image
    dir = os.path.join(os.getcwd(), savepath)
    if not os.path.exists(dir):
    os.makedirs(dir)
    filename = os.path.join(dir, basename % imgcounter)
     with open(filename, 'wb') as myfile:
           img_bytes = save_file(data, sock, size, myfile)
           # Converting bytes received into PIL image
            PIL_image = Image.open(io.BytesIO(img_bytes))
            bool_str = False
            # PREPROCESS IMAGE #
            input_image = preprocess_image(PIL_image,
input_shape, None)
             input_image2 = preprocess_image(PIL_image,
input_shape2, None)
            # PERFORM INFERENCE AND GET RESULT #
             result = perform_inference(input_image, exec_net,
input_blob, output_blob, 1)
             result2 = perform_inference(input_image2,
exec_net2, input_blob2, output_blob2, 1)

             result_message = RES_STRING + result
```

Figura 4.12 Ricezione dell'immagine, esecuzione inferenza e invio dei risultati.

Capitolo 5

Agricoltura di precisione

Durante lo sviluppo della tesi, si è affrontato il tema dell'agricoltura di precisione, una strategia aziendale volta ad informatizzare il settore agricolo cosicché ne vengano massimizzate le potenzialità e i frutti, minimizzando i danni ambientali. Si è dunque pensato a come la NCS2 sarebbe utile in tale settore, in particolare, nel caso della viticoltura, eseguendo una calibrazione di alcune reti neurali pre-addestrate sull'identificazione di grappoli d'uva e sulla localizzazione di foglie di vite affette da tre tipi di malattie (Isariopsis, mal dell'esca e marciume nero), convertendole nel formato IR e testandole con la NCS2.

5.1 Anaconda

Figura 5.1 Anaconda Logo. Fonte immagini e riferimenti: [48].

Anaconda è una distribuzione gratuita open source di Python, pensata, ad esempio, per lo sviluppo di programmi per il machine learning e data science, fornendo più di 1500 pacchetti, il pacchetto "conda" e un gestore di ambienti virtuali. Durante lo sviluppo della tesi è stata utilizzata tale distribuzione con la quale sono stati creati alcuni ambienti virtuali, installandovi i pacchetti necessari all'esecuzione delle applicazioni Python con *TensorFlow*.

5.2 Tensorflow

Come framework per l'utilizzo delle reti neurali si è scelto *TensorFlow*; le ragioni di tale scelta sono dovute alla sua compatibilità con la NCS2 e il toolkit OpenVINO™, alla moltitudine di modelli di reti neurali pre-addestrate forniti, all'enorme documentazione e alla community continuamente attiva.

Figura 5.2 TensorFlow logo. Fonte immagine e riferimenti: [49].

In particolare, TensorFlow è una libreria open source, sviluppata da *Google Brain*, per lo sviluppo di applicazioni di Machine Learning. È una seconda generazione di API ed è alla base di molteplici prodotti commerciali Google quali il riconoscimento vocale, Gmal, Google Foto e Ricerca. TensorFlow fornisce API native in linguaggio Python, C/C++, Java, Go e Rust. Nello sviluppo di questa tesi si sono utilizzate le API Python con le versioni 1.18.0, 1.15.2 e 1.13.1.

5.2.1 Object Detection API

TensorFlow Object Detection API [50] è un framework open source, presente sulla piattaforma GitHub, che si poggia su TensorFlow al fine di facilitare lo sviluppo, l'allenamento e infine l'impiego di modelli per Object Detection utilizzando, ad esempio, la tecnica del *Transfer Learning*, di cui in seguito. Nello sviluppo della tesi, tale framework è stato utilizzato sia sul PC a disposizione, sia su COLAB.

Il PC su cui è stato installato TensorFlow e le relative Object Detection API è composto dalle seguenti caratteristiche:

- Processore Intel® Core™ i7-6700 HQ 2.7Ghz;
- NVIDIA GeForce GTX 960m con 4GB di memoria;
- Sistema Operativo Windows 10 64 bit.

Per l'installazione su PC, si è creato un ambiente virtuale con Anaconda installandovi la versione Python 3.6.10 e il pacchetto "pip". Successivamente, si è installato la versione 1.13.1 di TensorFlow con

supporto per la GPU e infine una serie di pacchetti necessari per la compilazione di programmi e librerie.

La versione GPU di TensorFlow richiede la presenza di una scheda grafica NVIDIA, la versione del toolkit CUDA [51] di NVIDIA compatibile con la scheda grafica a disposizione, la 10.0 in questo caso, grazie al quale il codice parallelo è stato eseguito sulla GPU in questione, e infine, la cuDNN [52], libreria, anch'essa di NVIDIA, che fornisce un set di primitive per le reti neurali assieme a routine standard come la convoluzione, il pooling ecc. nella versione 7.4.2 per CUDA 10.0.

Fatto ciò, si è provveduto a scaricare la TensorFlow Object Detection API e compilare tutti i file nel formato ". proto" presenti all'interno così da generare il codice eseguibile delle librerie del framework. Vanno necessariamente compilati tutti i file di questo formato per il corretto funzionamento. Viene riportato, nella pagina successiva, il set di comandi che racchiude l'intera spiegazione.

- Aprire un prompt dei comandi di Anaconda con privilegi di amministratore e digitare i seguenti comandi:

```
C:\>conda create -n tensorflow1 pip python=3.6.10
C:\>activate tensorflow1
(tensorflow1) C:\>pip install tensorflow-gpu==1.13.1
(tensorflow1) C:\> conda install -c anaconda protobuf
(tensorflow1) C:\> pip install pillow
(tensorflow1) C:\> pip install lxml
(tensorflow1) C:\> pip install Cython
(tensorflow1) C:\> pip install contextlib2
(tensorflow1) C:\> pip install jupyter
(tensorflow1) C:\> pip install matplotlib
(tensorflow1) C:\> pip install pandas
(tensorflow1) C:\> pip install opencv-python
```

- Ora bisogna impostare le variabili d'ambiente, ciò va fatto ogni volta che il prompt dei comandi viene chiuso, a meno che non vengano memorizzate:

```
(tensorflow1) C:\> set PYTHONPATH =
C:\tensorflow1\models;C:\tensorflow1\models\research;C:\tensorflow1
\models\research\slim
```

- Spostarsi nella cartella "research"

```
(tensorflow1) C:\> cd C:\tensorflow1\models\research
```

- Compilare i file nel formato ".proto" presenti nella cartella ..\object_detection\protos con il seguente comando:

```
(tensorflow1) C:\>protoc --python_out=.
.\object_detection\protos\file_name.proto
```

- Ripetere il comando per ogni singolo file ".proto" e infine:

```
(tensorflow1) C:\tensorflow1\models\research> python
setup.py build
(tensorflow1) C:\tensorflow1\models\research> python
setup.py install
```

Per l'utilizzo del proprio Object Detector vi sono alcuni ulteriori passi da compiere: una volta che le Object Detection API sono state correttamente installate, si è scelto quale modello di rete neurale da allenare dal *TensorFlow detection model zoo* e si ha un dataset di immagini con i corrispondenti file".xml" - di cui verrà spiegato in seguito come generarli nel modo più intuitivo possibile - e non si ha un file CSV con le informazioni delle immagini organizzate nel seguente modo:

$$< nome\ del\ file\ immagine >, larghezza, altezza, classe, x_{min}, y_{min}, x_{max}, y_{max}$$

bisogna creare un eseguibile capace di trasformare i file ".xml" delle immagini in un file CSV organizzato in tale maniera e successivamente bisogna creare un ulteriore eseguibile che, a partire dai file CSV, generi i corrispondenti file in formato ".record", i quali saranno passati alla rete neurale per eseguire le relative fasi di addestramento e test. Una volta generati tutti i file necessari bisogna aprire la cartella *<tensorflow_installation_dir>/object_detection/samples/config* e copiare e incollare nel proprio progetto il file di configurazione relativo alla rete scelta; aprendo tale file si noterà dove vanno specificati i parametri da modificare per il corretto funzionamento.

Infine, va creato un file nel formato ".pbtxt" in cui si specificano gli oggetti da identificare associandovi un identificativo numerico univoco come nello schema seguente:

```
item{
  id: 1
  name: 'healty'
}

item{
  id: 2
  name: 'Esca'
}

…
```

Fatto ciò, nella cartella

<tensorflow_installation_dir>/object_detection/legacy

si possono trovare gli eseguibili Python *train.py* e *eval.py* necessari, rispettivamente, per eseguire la fase di addestramento e la fase di test della rete neurale.

5.2.2 TensorFlow detection model zoo

Il TensorFlow detection model zoo [53] fornisce un insieme di modelli pre-allenati su cinque dataset: il **COCO** dataset, il **Kitti** dataset, **l'Open Images** dataset, **l'AVA v2.1** dataset e l'**iNaturalist Species Detecion** Dataset.

Ogni modello scaricato contiene al suo interno:

- Un *graph proto* (graph.pbtxt);
- Un *checkpoint* composto dai file model.ckpt.data-00000-of-00001, model.ckpt.index, model.ckpt.meta;
- Un *frozen graph* del modello in questione da poter utilizzare per i test d'inferenza (frozen_inference_graph.pb);
- Un file di configurazione (pipeline.config) in cui si trovano le caratteristiche della modello.

Dall'insieme dei modelli, sono state scelte, da riaddestrare, la *Faster R-CNN InceptionV2* e la *SDD MobileNetV2*, entrambe pre-addestrate sul COCO dataset.

5.2.3 COCO dataset

Figura 5.3 COCO logo. Fonte immagine: [54].

COCO (Microsoft Common Object in Context) [55] è un dataset su vasta scala costruito per l'identificazione, la segmentazione e l'etichettatura di oggetti. Esso è composto da 330.000 immagini, di cui oltre 200.000 già etichettate, con 80 categorie di oggetti, 91 categorie di oggetti comuni, per un totale di circa 1,5 milioni di oggetti classificati. Tra i dataset proposti per l'Object Detection, quali IMAGENET, PASCAL VOC e SUN, è il più ampio in termini di dimensioni.

5.2.4 Transfer Learning

La scelta di riaddestrare i modelli è molto semplice: creare un modello da zero e addestrarlo oppure addestrare un modello già esistente da capo sono scelte che richiedono un'enorme potenza di calcolo e tempo. Inoltre, il dataset da utilizzare in questi casi richiederebbe che vi sia un gigantesco numero di categorie di oggetti che il sistema non deve, successivamente, riconoscere. Per tali ragioni, si è scelto di sfruttare la tecnica del *Transfer Learning*, ovvero sfruttare le conoscenze di un modello già addestrato su problema differente ma correlato. Ad esempio, un modello capace di riconoscere autovetture può essere addestrato per il riconoscimento di furgoni.

5.2.5 Reti neurali utilizzate

Come anticipato nel paragrafo 5.2.2, sono state estratte dal TensorFlow detection model zoo la:

- **Faster R-CNN InceptionV2**: tale modello combina la **Faster R-CNN**, spiegata nel Capitolo 3, con la **InceptionNetV2** [56], una rete creata per la classificazione di immagini e molto accurata nella fase di inferenza, per via dell'utilizzo di layer completamente connessi, a discapito però della velocità d'esecuzione. La caratteristica principale di questa rete è quella di essere più ampia che profonda. Si è scelto questo modello principalmente per il rapporto velocità/mAP dichiarato dagli sviluppatori di TensorFlow;
- **SSD MobileNetV2**: come si evince dal nome, questo modello è l'unione di SSD, spiegato anch'esso nel Capitolo 3, e della MobileNetV2 [57]. La MobileNet è una rete progettata per essere eseguita, principalmente, su dispositivi mobili o dalla bassa capacità computazionale (e.g. Raspberry Pi) così da garantirne portabilità e velocità d'esecuzione a discapito, però, dell'accuratezza generale in fase d'inferenza. Di base, anche questa è una rete neurale volta alla Image Classification ma con l'applicazione dell'SSD detector, è stata convertita all'Object Detection. Questo modello, invece, è stato scelto per testarne l'effettiva velocità d'inferenza sulla NCS2.

Infine, vi è un'altra rete di cui si è voluto testare la potenzialità, la **YOLOv3**, ma di cui TensorFlow non offre alcun modello pre-addestrato; pertanto, si è utilizzato un progetto GitHub [58], che ne conteneva la topologia scritta in codice Python con l'utilizzo di TensorFlow, scaricando i pesi e i bias pre-addestrati sempre sul COCO dataset dal sito degli sviluppatori del detector [59] e, infine, è stato eseguito il Transfer Learning con un eseguibile Python incluso, anch'esso, nel progetto GitHub sopracitato.

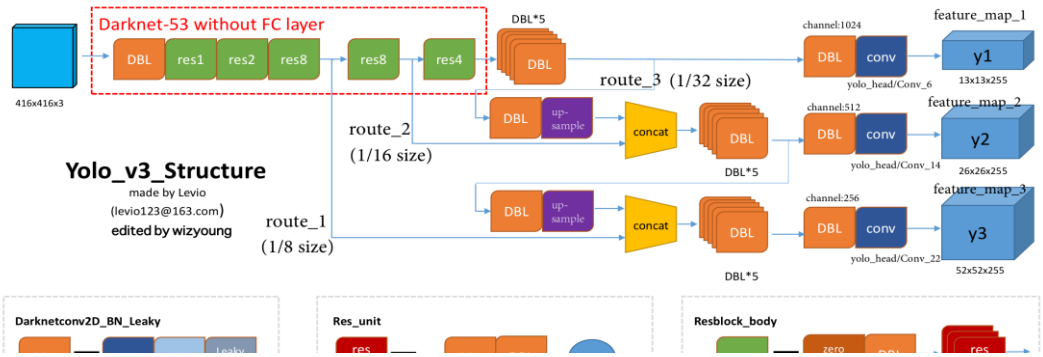


Figura 5.4 Struttura della YOLOv3 utilizzata.

5.3 LabelImg

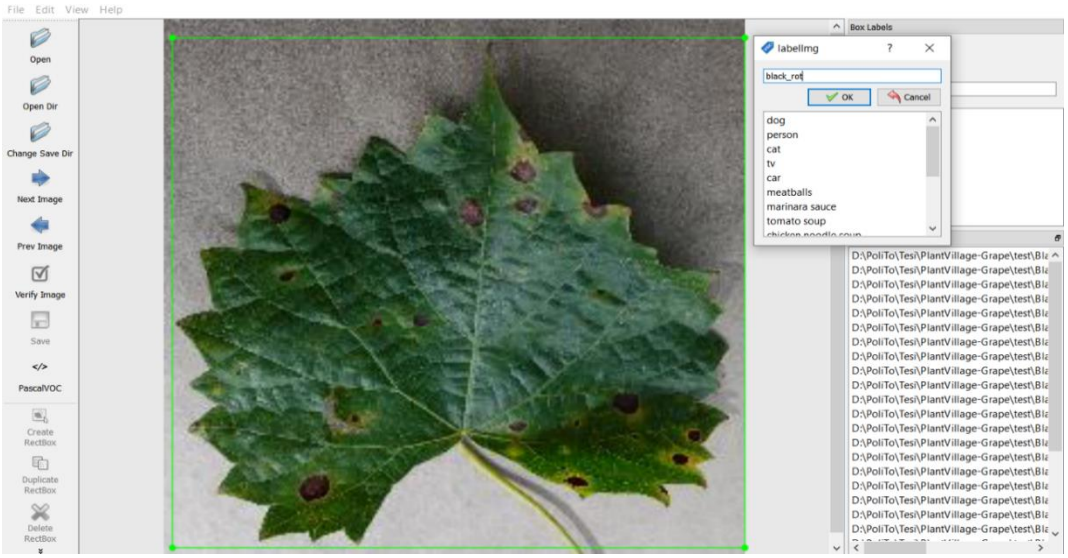


Figura 5.5 Esempio di esecuzione di LabelImg con foglia di vite.

TensorFlow richiede che, per ri-addestrare un modello, insieme al dataset di immagini vengano forniti i file ".xml" per ogni immagine, contenenti le dimensioni dell'immagine e la lista degli oggetti presenti nell'immagine stessa, aventi come parametri l'etichetta associata e le coordinate che delimitano il rettangolo al cui interno è presente l'oggetto da identificare. Per fare ciò, nella creazione del dataset delle foglie di vite è stato usato un potente progetto GitHub chiamato *LabelImg* [60], con il quale basta aprire la cartella contente le immagini interessate, disegnare con il puntatore del mouse i box attorno agli oggetti da identificare, specificarne l'etichetta e salvare il file ".xml" associato all'immagine appena processata.

5.4 Google Colaboratory

Figura 5.6 Google Colaboratory logo. Fonte immagine e riferimenti: [61].

Durante l'esecuzione del Transfer Learning per il riconoscimento dei grappoli d'uva, sono state messe in luce le limitazioni del computer a disposizione per quanto riguarda un ri-addestramento ragionevole delle reti: sono state impiegate circa sette ore per un ri-addestramento di 10.000 step per la SSD MobileNetV2 e la Faster R-CNN InceptionV2 e altrettanto tempo per ri-addestrare la YOLOv3 per 500 epoche.

Per tali ragioni, spostando la ricerca sull'identificatore di foglie di vite malate, si è utilizzato **Google Colaboratory** o, più semplicemente, **Colab**.

```
[ ] !python eval.py --logtostderr --pipeline_config_path=ssd_mobilenet_v2_coco.config --checkpoint_dir=training_mobilenet/ --eval_dir=training_mobilenet/eval_1

[ ] !python export_inference_graph.py --input_type image_tensor --pipeline_config_path ssd_mobilenet_v2_coco.config --trained_checkpoint_prefix training_mobilenet/model.ckpt-5000 --output_direct

[ ] %cd /content/gdrive/My Drive/Training Model test/models/research/object_detection/

    import numpy as np
    import os
    import six.moves.urllib as urllib
    import sys
    import tarfile
    import tensorflow as tf
    import zipfile
    from distutils.version import StrictVersion
    from collections import defaultdict
    from io import StringIO
    from matplotlib import pyplot as plt
    %matplotlib inline
    from PIL import Image
    # This is needed since the notebook is stored in the object_detection folder.
    sys.path.append("..")
    from object_detection.utils import ops as utils_ops
    from object_detection.utils import label_map_util
    from object_detection.utils import visualization_utils as vis_util

    # What model to download.
    MODEL_NAME = '/content/gdrive/My Drive/Training Model test/models/research/object_detection/Leaf_Desease_Tensorflow/My_exp_graph'

    # Path to frozen detection graph. This is the actual model that is used for the object detection.
    PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'
```

Figura 5.7 Esempio di divisione del codice in celle in Colab.

Colab è una piattaforma che permette di eseguire codice su cloud sfruttando i Jupyter Notebook, dei documenti interattivi nei quali si possono scrivere righe di codice eseguibile o comandi da lanciare, il tutto organizzato in celle.

Tale piattaforma fornisce agli utenti una GPU Tesla K80 per il training e il test dei propri algoritmi di Machine Learning e Deep Learning assieme al supporto di Python dalla versione 2.7 alla 3.6 con varie librerie quali TensorFlow, Keras, PyTorch e OpenCV.

L'utilizzo di Colab ha velocizzato enormemente il lavoro, permettendo di avere, con un ri-addestramento di circa sei ore, i modelli di Faster R-CNN InceptionV2 e SSD MobileNetV2 allenati per 50.000 step.

5.5 Localizzazione dei grappoli d'uva

Come primo approccio al Transfer Learning, sono state riaddestrate le reti neurali citate nel paragrafo 5.2.5 sull'identificazione dei grappoli d'uva.

5.5.1 Open Images Dataset

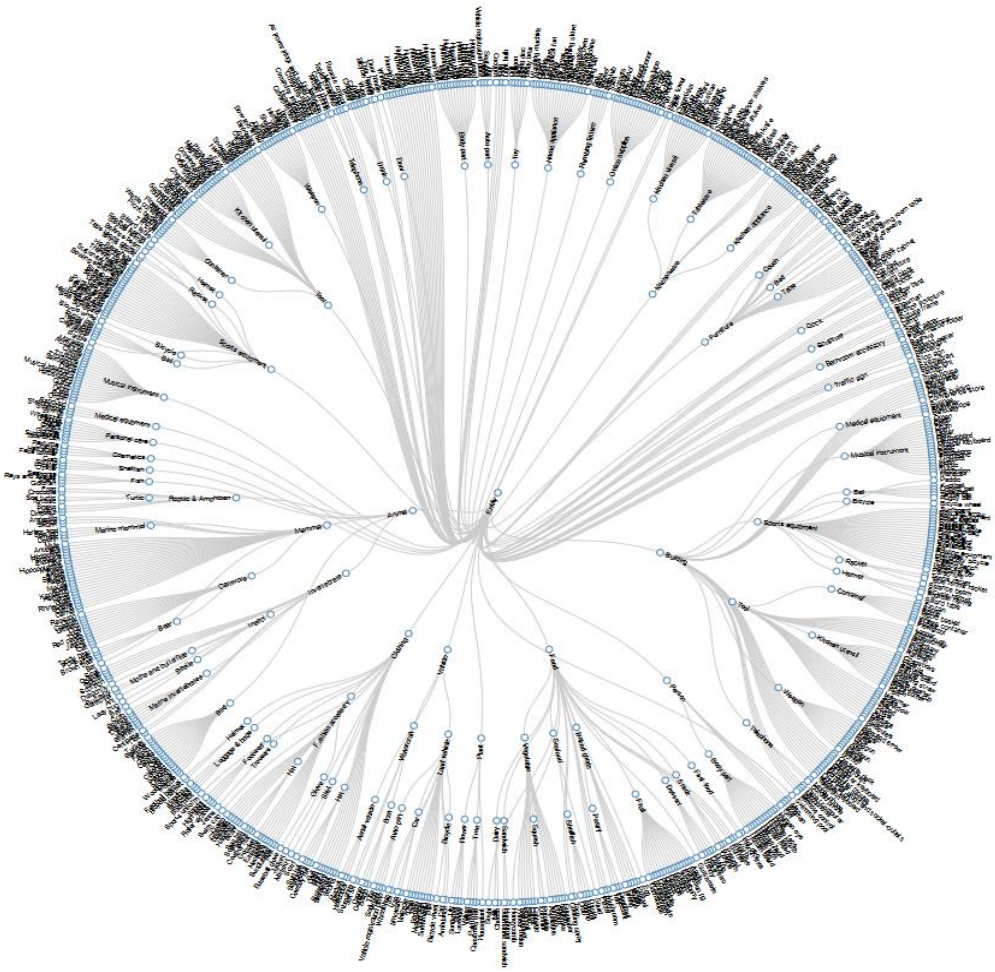


Figura 5.8 Distribuzione delle gerarchie nell'Open Images dataset. Fonte immagine e riferimenti: [62].

Le immagini dei grappoli d'uva utilizzate per il training e il testing dei modelli sono state estratte dall'*Open Images Dataset v4* (OIDv4), un dataset di circa 9,2 milioni di immagini, tutte sotto licenza *Creative Commons Attribution*, con le relative annotazioni contenenti etichette e coordinate dei box identificatori, così da facilitarne l'utilizzo nell'impiego dell'Image Classification e dell'Object Detection. In particolare, per l'Object Detection, si hanno 1,9 milioni di immagini con 15,4 milioni di box, quest'ultimi disegnati a mano da annotatori professionisti garantendo accuratezza e consistenza. Inoltre, è già presente una suddivisione tra insiemi di training, validation e test.

Tuttavia, il problema di tale dataset è che non si può estrarre solo la classe di immagini interessata ma bisogna necessariamente scaricarlo interamente. Per ovviare a ciò, si è utilizzato un progetto GitHub denominato *OIDv4 ToolKit* [63], che ha permesso di scaricare dall'OIDv4 tutte le immagini della categoria "Grape" organizzate nelle cartelle training, test e validation, ognuna contenente un'ulteriore cartella denominata "Label" con una lista di file ".txt", con le coordinate dei box per ogni elemento presente in una foto, aventi la seguente struttura:

<nome_classe> x_min y_min x_max y_max

Successivamente, per generare i file ".xml" richiesti si è sviluppato un eseguibile Python che legge ogni file ".txt" associato ad un'immagine e ne crea il file ".xml" associato.

```
# PATH TO LABELS FOLDER
labels_foder =
r"C:\tensorflow1\models\research\object_detection\Grape-Detection-
Using-Tensorflow\images\test\Label"
labels_path = Path(labels_foder)
# PATH TO IMAGES FOLDER
images_folder =
r"C:\tensorflow1\models\research\object_detection\Grape-Detection-
Using-Tensorflow\images\test"
images_path = Path(images_folder)

for filename in os.listdir(labels_foder):
    if filename.endswith(".txt"):
        with open(os.path.join(labels_foder, filename), 'r') as f:
            annotation_list = f.readlines()
            image_name = filename.split(".")[0]
        image_file = images_folder + '\\' + image_name + ".jpg"

        xmlConvert(image_file, annotation_list)
```

Figura 5.9 Definizione e inizializzazione dei parametri e richiamo della funzione di conversione.

```
def xmlConvert(image_path, annotation_list):
    image_path = Path(image_path)
    img = np.array(Image.open(image_path).convert('RGB'))

    #DEFINE LABELIMG XML FORMAT
    annotation = ET.Element('annotation')
    ET.SubElement(annotation, 'folder').text = str(image_path.parent.name)
    ET.SubElement(annotation, 'filename').text = str(image_path.name)
    ET.SubElement(annotation, 'path').text = str(image_path)

    source = ET.SubElement(annotation, 'source')
    ET.SubElement(source, 'database').text = 'Unknown'

    size = ET.SubElement(annotation, 'size')
    ET.SubElement(size, 'width').text = str(img.shape[1])
    ET.SubElement(size, 'height').text = str(img.shape[0])
    ET.SubElement(size, 'depth').text = str(img.shape[2])

    ET.SubElement(annotation, 'segmented').text = '0'

    for annot in annotation_list:
        tmp_annot = annot.split(' ')
        cords, label = tmp_annot[1:5], tmp_annot[0]
        xmin, ymin, xmax, ymax = float(cords[0]), float(cords[1]),
float(cords[2]), float(cords[3])

        object = ET.SubElement(annotation, 'object')
        ET.SubElement(object, 'name').text = label
        ET.SubElement(object, 'pose').text = 'Unspecified'
        ET.SubElement(object, 'truncated').text = '0'
        ET.SubElement(object, 'difficult').text = '0'

        bndbox = ET.SubElement(object, 'bndbox')
        ET.SubElement(bndbox, 'xmin').text = str(int(xmin)).strip()
        ET.SubElement(bndbox, 'ymin').text = str(int(ymin)).strip()
        ET.SubElement(bndbox, 'xmax').text = str(int(xmax)).strip()
        ET.SubElement(bndbox, 'ymax').text = str(int(ymax)).strip()

    tree = ET.ElementTree(annotation)

    xml_file_name = image_path.parent / (image_path.name.split('.')[0] +
'.xml')
    with open(xml_file_name, "wb") as f:
        tree.write(f, pretty_print=True, xml_declaration=True, encoding="utf-
8")
```

Figura 5.10 Creazione e scrittura del file .xml strutturato nel formato LabelImg.

Infine, il dataset utilizzato è composto da 767 immagini per la fase di training e 122 per la fase di testing.

5.5.2 Risultati ottenuti

Sono riportati, di seguito, i risultati ottenuti durante le fasi di training e testing. Come già anticipato nel paragrafo 5.4, questo primo approccio al Transfer Learning è stato compiuto sul computer a disposizione; pertanto, i risultati, anche se discreti, possono essere nettamente migliorabili con l'utilizzo di una macchina più potente o di Google Colaboratory.

SSD MobileNetV2

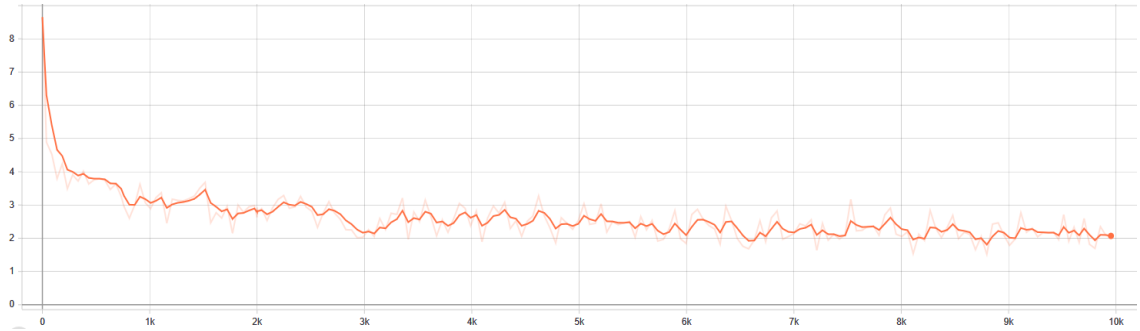


Figura 5.11 Perdita in fase di training.

Analizzando il grafico in figura 5.11, durante la fase di training la perdita ha un valore alla prima iterazione di 8,7 per poi decrementare, dopo circa 1000 iterazioni, al valore di 2,9. Per la SSD MobileNetV2, una perdita accettabile si comincia ad avere da valori inferiori a 2, raggiunto dopo circa 10.000 step di training. Per un training più efficace, il valore di step da raggiungere sarebbe di circa 40.000; dunque, è ragionevole credere che si possano raggiungere risultati ancora più soddisfacenti.

	IoU = 0.50:0.95	**IoU = 0.50**	**IoU = 0.75**
Average Precision	0,231	0,391	0,245

Tabella 5.1 Valori di AP per SSD MobileNetV2.

Analizzando la precisione media sul test set, possiamo notare che questa non è molto alta, circa 0,4 quando il valore di IoU è di 0,50. Un risultato abbastanza prevedibile considerando che il modello di rete neurale in questione non è stato allenato a sufficienza ma, soprattutto, non è famoso per le sue performance in termini di accuratezza.

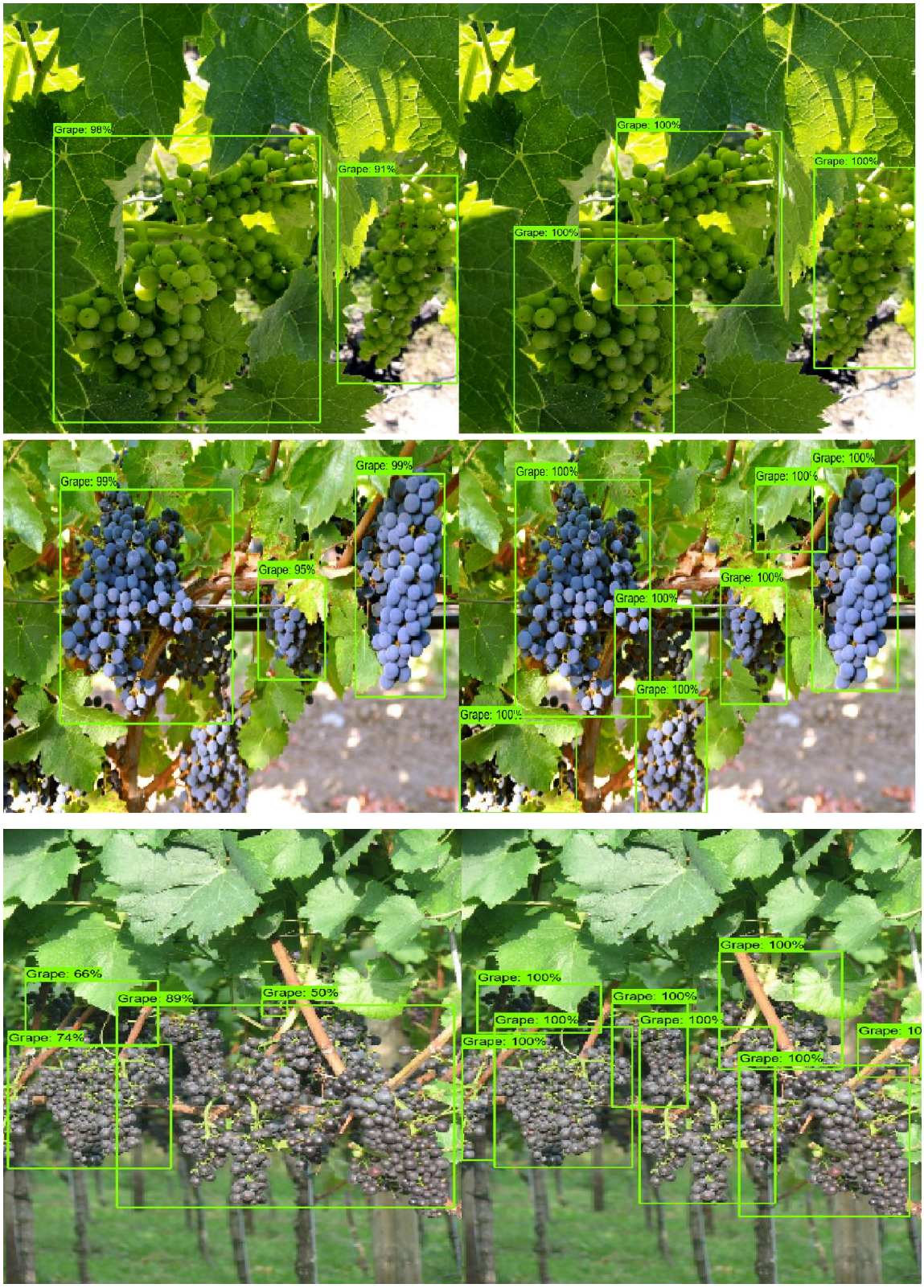


Figura 5.12 Identificazione dei grappoli per SSD MobileNetV2. A destra l'identificazione correttta, a sinistra quella eseguita dalla rete.

Faster R-CNN InceptionV2

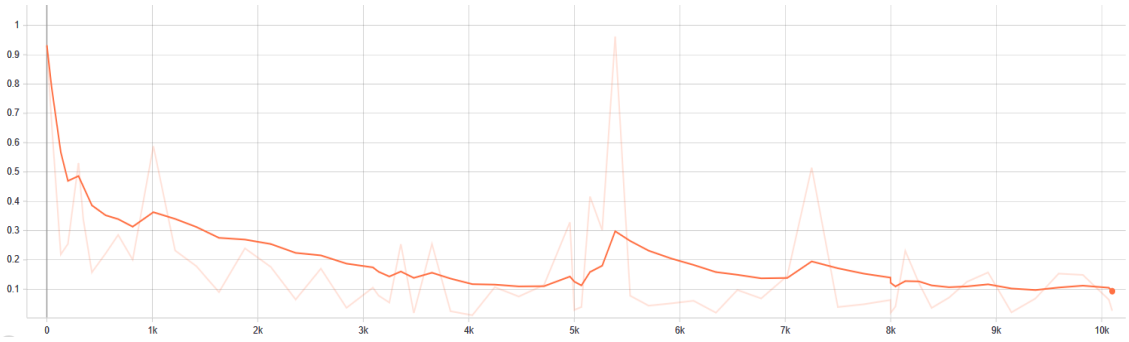


Figura 5.13 Perdita in fase di training per Faster R-CNN InceptionV2.

Possiamo subito notare che la perdita ha un valore di 0,9 già dalla prima iterazione, a differenza della SSD MobileNetV2. Inoltre, dopo 9000 step questa inizia a stabilizzarsi su valori attorno allo 0,1; dopo 10.000, invece, si nota come la perdita scende anche a valori sullo 0,09.

Si deduce che la Faster R-CNN InceptionV2 ottenga valori decisamente migliori rispetto alla precedente e che potrebbero aumentare con il proseguimento del training.

	IoU = 0.50:0.95	**IoU = 0.50**	**IoU = 0.75**
Average Precision	0,278	0,443	0,296

Tabella 5.2. Valori di AP per Faster R-CNN InceptionV2.

Si può notare, inoltre, come anche i valori di precisione media siano più alti rispetto alla rete neurale precedente. Ciò mette in luce come tale modello di rete neurale sia stata pensata più con l'obiettivo dell'accuratezza che della velocità d'inferenza.

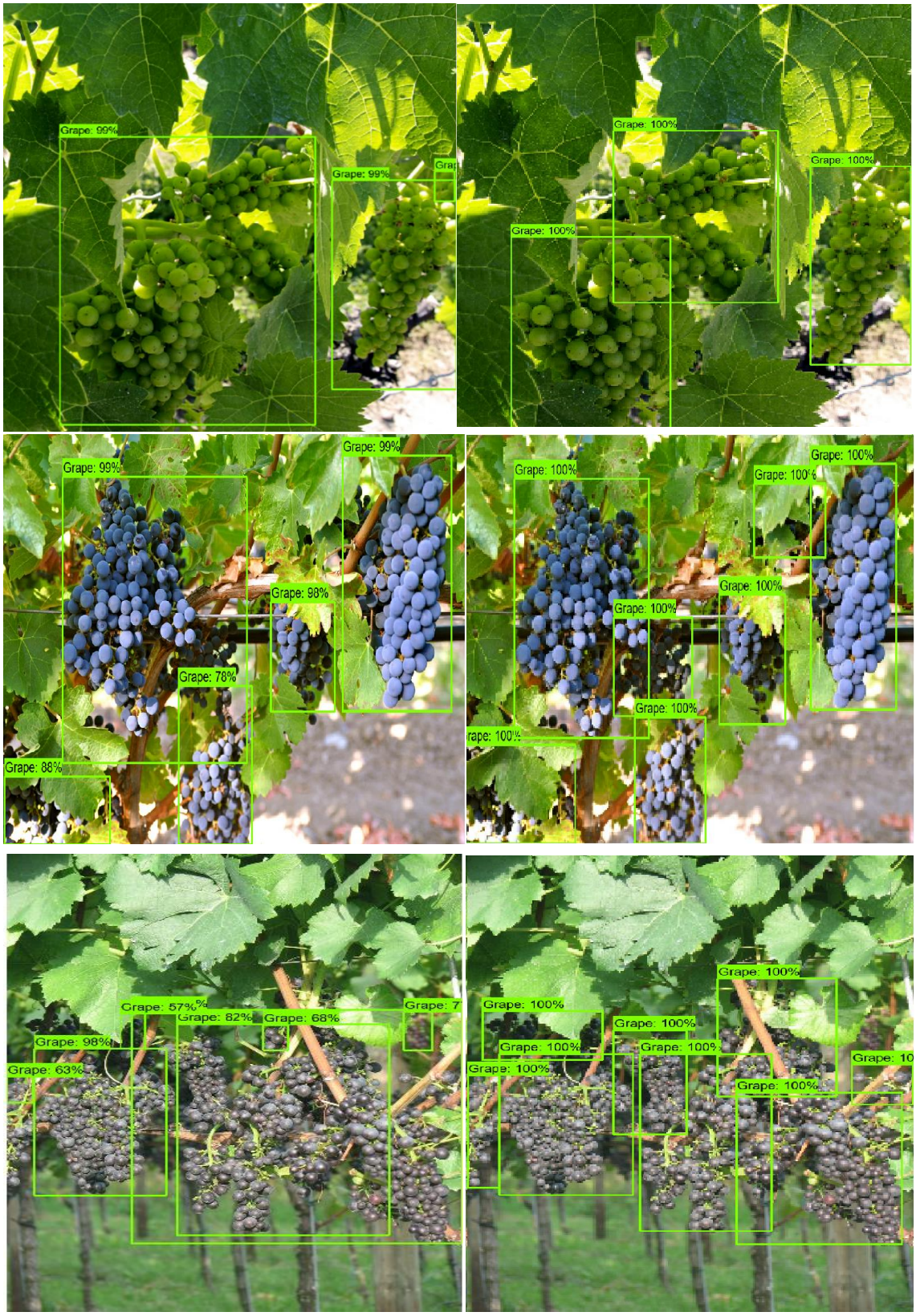


Figura 5.14 Identificazione dei grappoli per Faster R-CNN InceptionV2.

YOLOv3

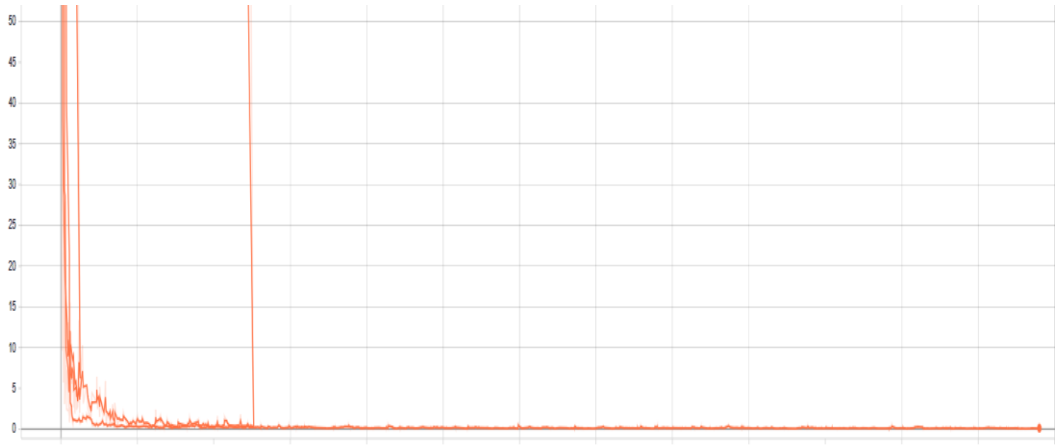


Figura 5.15 Perdita in fase di training per YOLOv3.

Come possiamo notare dal grafico, la perdita ha un valore molto alto alle prime iterazioni ma discende rapidamente fino a stabilizzarsi per valori di circa 0,1.

Figura 5.16 AP in fase di testing per ogni epoca.

La precisione media si attesta su valore di circa il 33%, non molto alto, ci si aspettava di più. La linea retta, che si può notare in entrambi i grafici, è dovuta ad una fase di training eseguita in molteplici slot di tempo; ciò ha comportato un leggero squilibrio nell'andamento temporale del grafico, poiché il progetto GitHub di questa YOLO memorizzava i checkpoint solo dopo aver eseguito un certo numero di epoche.

Figura 5.17 Identificazione dei grappoli per YOLOv3.

Considerazioni personali sui risultati conseguiti

Nonostante la fase di training abbia effettuato un numero esiguo di step, nel testing delle prime due reti su alcune immagini, il numero di True Positive sembra essere abbastanza soddisfacente.

Essendo i modelli della SSD MobileNetV2 e della Faster R-CNN InceptionV2 forniti dagli sviluppatori di TensorFlow, i risultati ottenuti sono quelli aspettati, a differenza della YOLOv3 in cui si è utilizzata un'implementazione in puro TensorFlow che, tuttavia, non ha fornito i risultati sperati, anzi, contrariamente a quanto ci si aspettava, si è comportata decisamente peggio delle altre. Inoltre, OpenVINO™ fornisce dei file di configurazione per la conversione corretta, nel formato IR, dei modelli addestrati con la TensorFlow object detection API; per tali ragioni, si è deciso di eseguire la conversione nel formato IR e l'addestramento delle reti per le foglie di vite, i cui risultati al paragrafo 5.6, soltanto per la Faster R-CNN InceptionV2 e la SSD MobileNetV2.

5.5.3 Confronto con NCS2

SSD MobileNet v2

Per il corretto funzionamento del modello della rete neurale in questione sulla NCS2, bisogna utilizzare il Model Optimizer con il supporto di un file in formato "*.json*", fornito anch'esso con il toolkit, il quale si può trovare al path:

<INSTALL_DIR>\deployment_tools\model_optimizer\extensions\front\tf

e prende il nome di "*ssd_v2_support.json*". Questo tipo di file è necessario perché descrive le regole per la conversione di specifiche topologie di modelli TensorFlow.

Osserviamo, nella tabella seguente, la valutazione del modello con la benchmark_app di OpenVINO™ per la CPU e la NCS2.

benchmark_app	**#Iterazioni**	**Latenza(ms)**	**Throughput(fps)**
CPU	9588	24,34	159,69
NCS2	2628	91,53	43,71

Tabella 5.3 Risultati della benchmark_app per la SSD Mobile NetV2 fornendo quattro immagini diverse per l'inferenza.

Come previsto, la CPU ha prestazioni d'inferenza nettamente migliori della NCS2; tuttavia, sia latenza che throughput sono valori decisamente accettabili per un utilizzo della NCS2 su campo. Vediamo ora, utilizzando le foto riportate in figura 5.17 e un altro programma fornito con OpenVINO™, "*Object_detection_sample_ssd*", volto al testing dei modelli per Object Detection convertiti nel formato IR, il risultato dell'identificazione con il dispositivo.

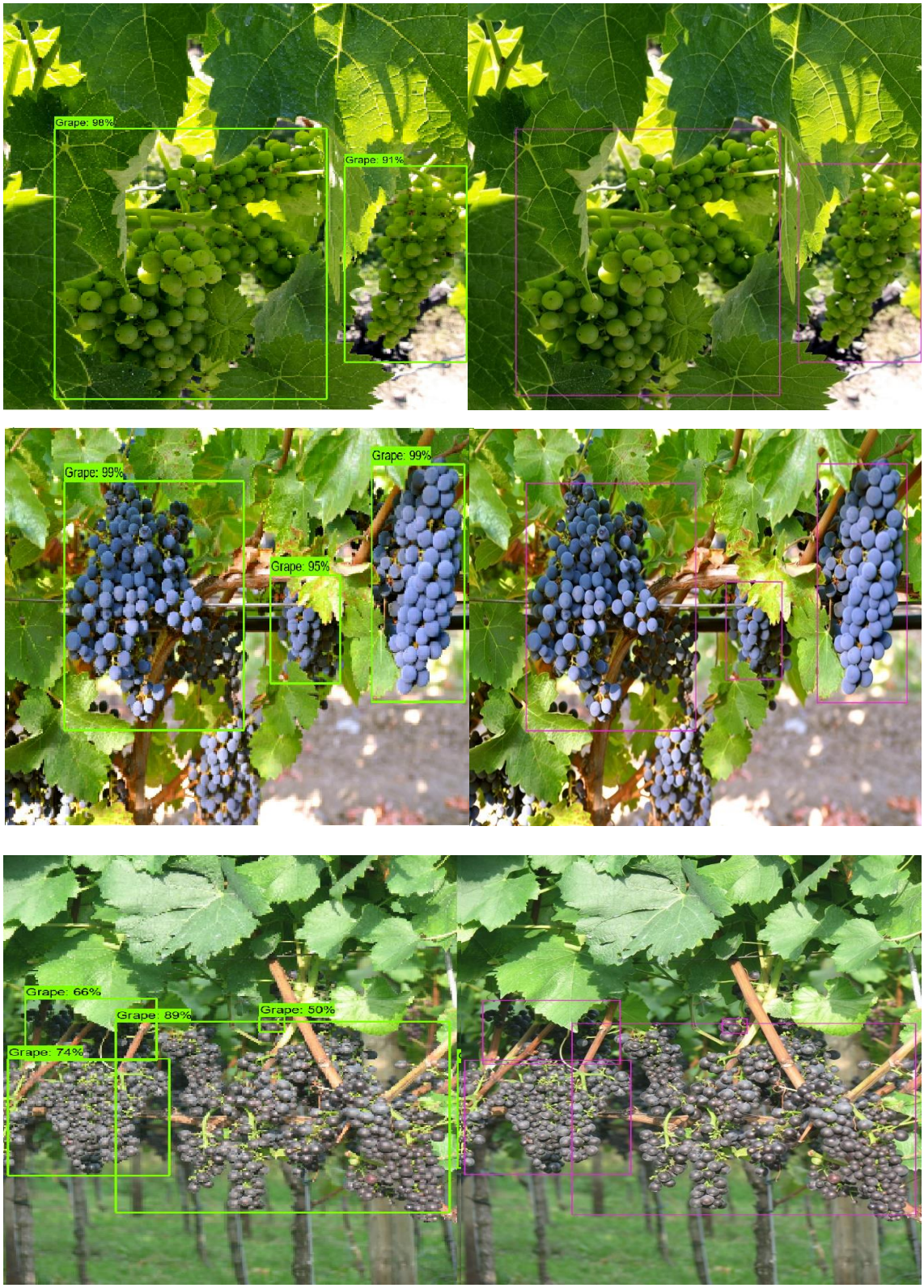


Figura 5.18 Inferenza con NCS2 e SSD MobileNetV2.

Come possiamo osservare in figura 5.18, la rete neurale ha identificato i grappoli esattamente come durante la fase di testing, evidenziando come il processo di conversione non abbia portato ad alcuna perdita di precisione.

Sono riportati di seguito, invece, i parametri registrati nell'utilizzo del modello, caricato sulla NCS2, con la webcam del computer, utilizzando un'altra applicazione fornita con OpenVINO™ denominata "*object_detection_demo_ssd_async*".

object_detection_demo_ssd_async	**Detection time (ms)**	**Detection time (fps)**
CPU	~7	~130
NCS2	~44	~23

Tabella 5.4 Risultati Object_detection_demo_ssd_async con SSD MobileNetV2 utilizzando la webcam.

La CPU ottiene ancora performance migliori; tuttavia, 23fps sono un valore abbastanza ragionevole per ottenere un'identificazione fluida delle immagini acquisite.

Figura 5.19 Cattura con la webcam, NCS2 e SSD MobileNetV2 visualizzando un'immagine da cellulare.

Faster R-CNN InceptionV2

Come nel caso precedente, per la corretta conversione del modello è stato utilizzato il file di configurazione *faster_rcnn_support_api_v1.14.json*. Si notano già i primi problemi tra questa rete e la SSD MobileNetV2 al caricamento del modello sulla chiavetta, in quanto sono richiesti ben cinque minuti, al contrario del caricamento sulla CPU, il quale è stato istantaneo; il problema in questione, come riportato sul forum Intel (https://software.intel.com/en-us/forums/intel-distribution-of-openvino-toolkit/topic/805411), pare essere un baco nel caricamento del modello sul dispositivo e, nonostante il problema sia stato sollevato nel 2019, ancora non è stata fornita una soluzione. Analizziamo ora i dati della *benchmark_app*:

benchmark_app	**#Iterazioni**	**Latenza(ms)**	**Throughput(fps)**
CPU	176	1398,83	2,84
NCS2	52	5338,12	0,5

Tabella 5.5 Risultati di benchmark_app per Faster R-CNN Inception v2.

Notiamo subito come, durante la fase di realizzazione, l'obiettivo della Faster R-CNN InceptionV2 non era la velocità d'inferenza: la latenza supera 1s e il throughput ha un valore decisamente basso, 2,84 fps, rispetto alla rete precedente, tutto ciò utilizzando la CPU. Prestazioni ancora più basse si hanno con la NCS2; dunque, si può affermare che la rete in questione richiederebbe un dispositivo molto più potente per un funzionamento soddisfacente in termini di velocità. Vediamo ora come il modello convertito, utilizzando la NCS2, classifica le tre immagini.

Figura 5.20 Inferenza con NCS2 e Faster R-CNN InceptionV2.

Anche in questo caso, il modello si è comportato esattamente come in fase di testing; da notare, inoltre, come nella prima immagine della griglia, i box coincidano perfettamente con quelli ground-truth riportati in figura 5.14.

Vengono riportati, di seguito, le statistiche d'inferenza del modello con la webcam:

object_detection_demo_ssd_async	**Detection time (ms)**	**Detection time (fps)**
CPU	~460	~2,30
NCS2	~1530	~0,66

Tabella 5.6 Risultati Object_detection_demo_ssd_async con Faster RCNN Inception v2 utilizzando la webcam.

Dai valori riportati nella tabella 5.6, si evince come, anche nell'utilizzo della webcam, il modello sia molto lento nell'esecuzione dell'inferenza.

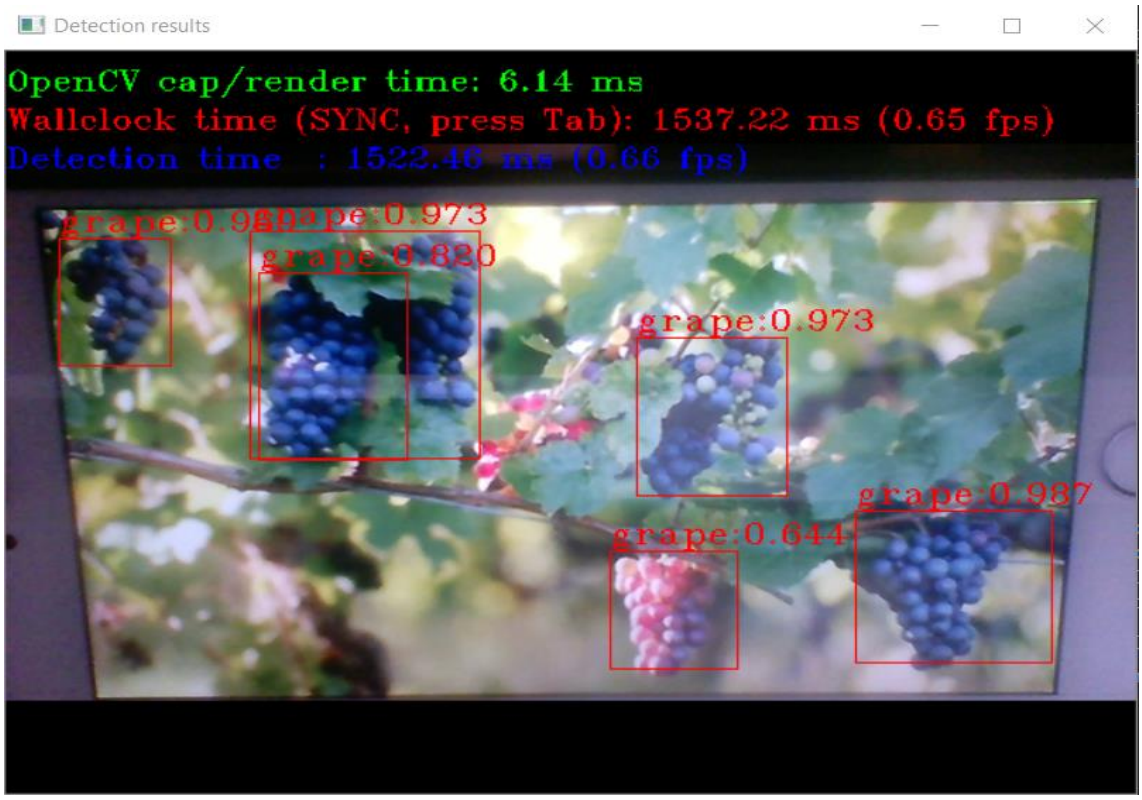


Figura 5.21 Cattura con la webcam, NCS2 e Faster RCNN InceptionV2 visualizzando un'immagine da cellulare.

Tuttavia, come ci si aspettava, le prestazioni per quanto riguarda l'accuratezza, sono molto più soddisfacenti della rete precedente.

5.6 Identificazione di malattie nelle foglie di vite

Successivamente al tema dell'identificazione dei grappoli d'uva, si è proseguito il lavoro di tesi con un tema molto recente riguardante l'agricoltura di precisione, ovvero l'identificazione di foglie di vite malate. In questo caso, per le ragioni spiegate nel paragrafo 5.5, si è preferito concentrarsi sull'addestramento delle sole Faster R-CNN InceptionV2 e SSD MobileNetV2.

5.6.1 PlantVillage Dataset

Figura 5.22 PlantVillage logo. Fonte immagine: [64].

Il dataset utilizzato durante questa parte del lavoro di tesi è il PlantVillage dataset [65] che consiste in circa 55.000 immagini di foglie sane e malate, divise per 38 categorie di specie e malattie. L'idea nacque nel 2015 con David P.Hughes e Marcel Salathé con lo scopo di sviluppare un sistema mobile capace di eseguire una diagnostica delle malattie delle piante. In passato vi era accesso libero ma oggi non più; tuttavia, è possibile trovare su Internet qualche versione meno recente del dataset in questione. Quella utilizzata per la tesi è stata reperita un progetto GitHub [66]. Da tale dataset sono state estratte le immagini delle foglie di vite, di cui esistono quattro categorie: healty (sane), black_rot (foglie affette dal marciume nero), esca (malattia

dell'esca) e isariopsis (un tipo di fungo della foglia di vite). Essendo il dataset sprovvisto delle annotazioni per ogni foto, sono stati creati i file ".xml" con LabelImg, ottenendo un sottoinsieme del dataset, diviso in immagini di training e di test, strutturato nel seguente modo:

- Training set: 465 immagini
  - Foglie sane (healty): 276 immagini;
  - Foglie malate: 189 divise in:
    - Black_rot: 56 immagini;
    - Esca: 67 immagini;
    - Isariopsis: 64 immagini;
- Test set: 99 immagini, circa 23 per ogni categoria.

Le immagini presenti nel dataset ritraggono le foglie in primo piano, sia per il training set che per il test set. Come anticipato nel paragrafo 5.4, il training è stato eseguito su Google Colaboratory per un totale di 50.000 step per entrambe le reti.

5.6.2 Risultati ottenuti

SSD MobileNetV2

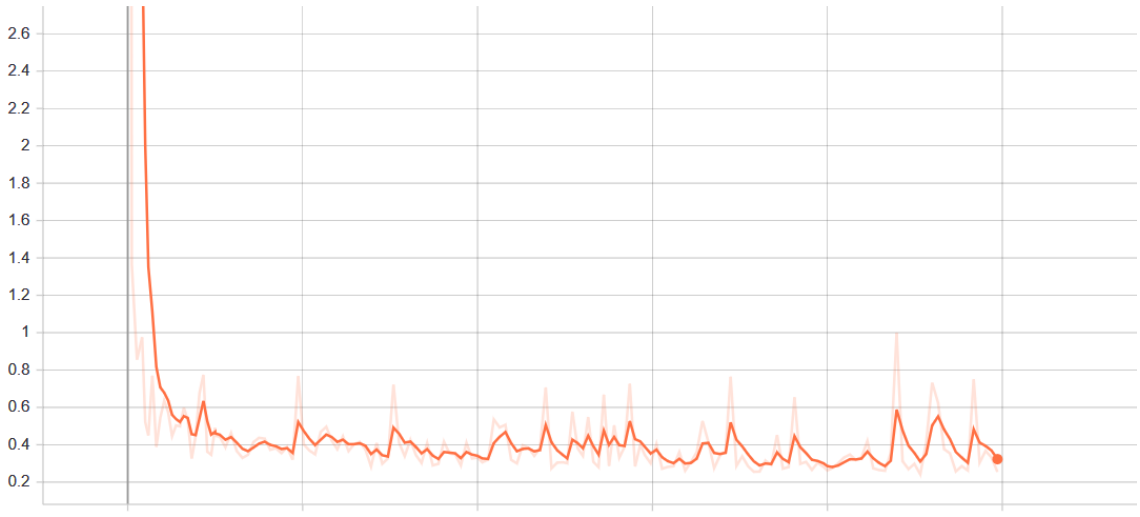


Figura 5.23 Perdita in fase di training.

Come nel caso precedente, la perdita della SSD MobileNetV2 alle prime iterazioni è molto alta, infatti allo step 527 il valore è di 4,9 per poi

cominciare a stabilizzarsi attorno allo 0,3 dopo 10.000 step d'esecuzione, infatti, allo step finale, il valore di perdita è pari a 0,323.

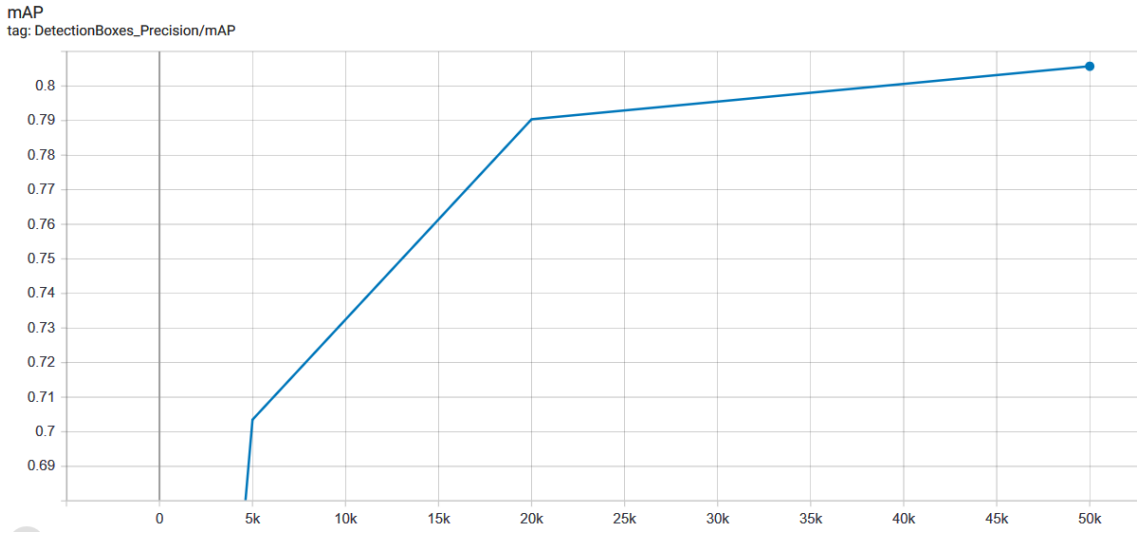
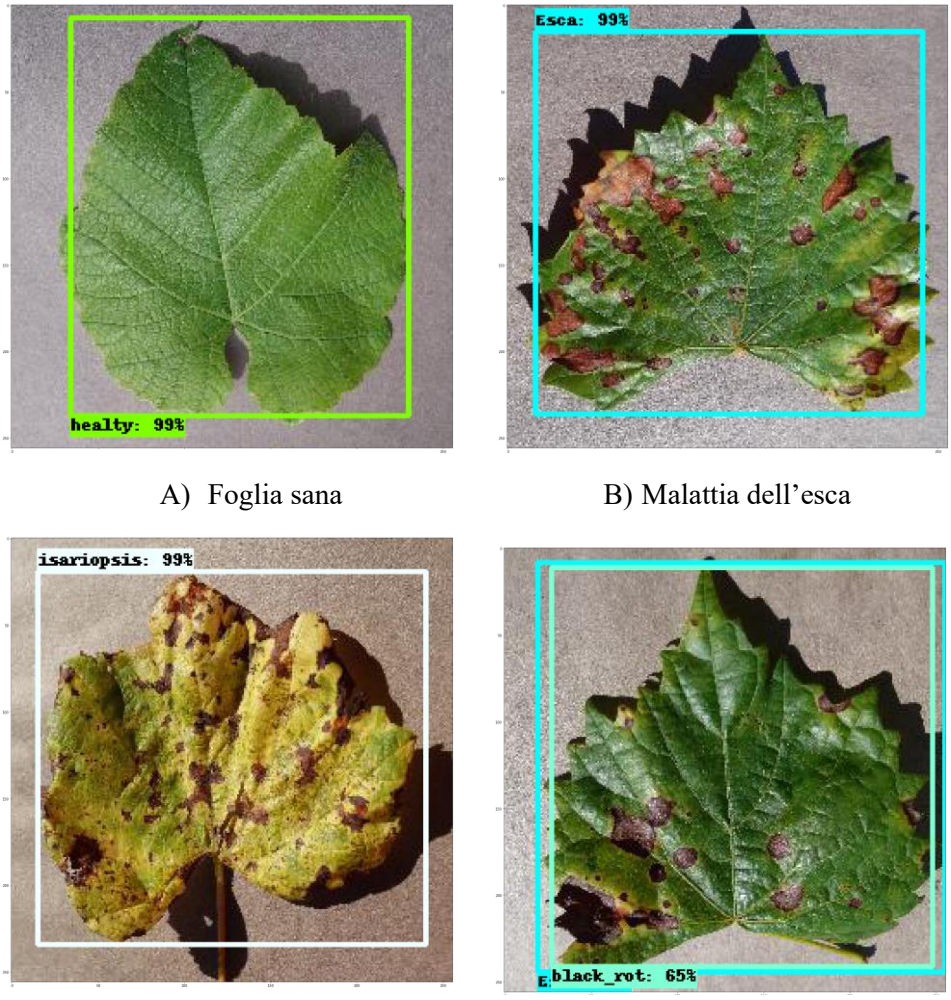


Figura 5.24 mAP per SSD MobileNet v2.

Notiamo invece come il valore di mAP sul test set tenda ad aumentare con il passare del tempo. Si è registrato un valore di mAP di 0,7 allo step 5000, 0,79 allo step 20.000 e infine 0,81 allo step 50.000.

Analizziamo ora i risultati della fase di inferenza su una singola immagine per ognuna delle quattro categorie:



A) Foglia sana

B) Malattia dell'esca

C) Isariopsis

D) Marciume nero

Figura 5.25 Inferenza su foglie di vite con SSD MobileNet v2.

Possiamo notare come la rete neurale abbia riconosciuto correttamente le malattie delle foglie, mostrando qualche incertezza sul marciume nero, infatti, osservando la foto D), si possono notare i box con etichetta "black_rot" ed "Esca" che si sovrappongono.

Faster R-CNN InceptionV2

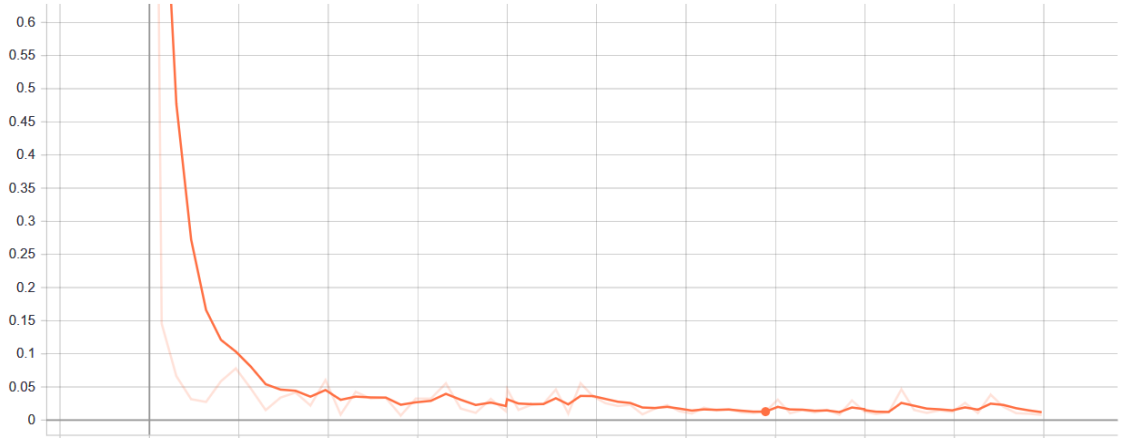


Figura 5.26 Perdita in fase di training.

Possiamo subito notare come la Faster R-CNN InceptionV2, durante il training, assuma dei valori di perdita decisamente inferiori già dai primi step, rispetto alla SSD MobileNetV2; infatti, dopo 1000 step il valore di perdita è già di 0,02, dopo 30.000 step ci si comincia a stabilizzare attorno a valori prossimi allo 0,01.

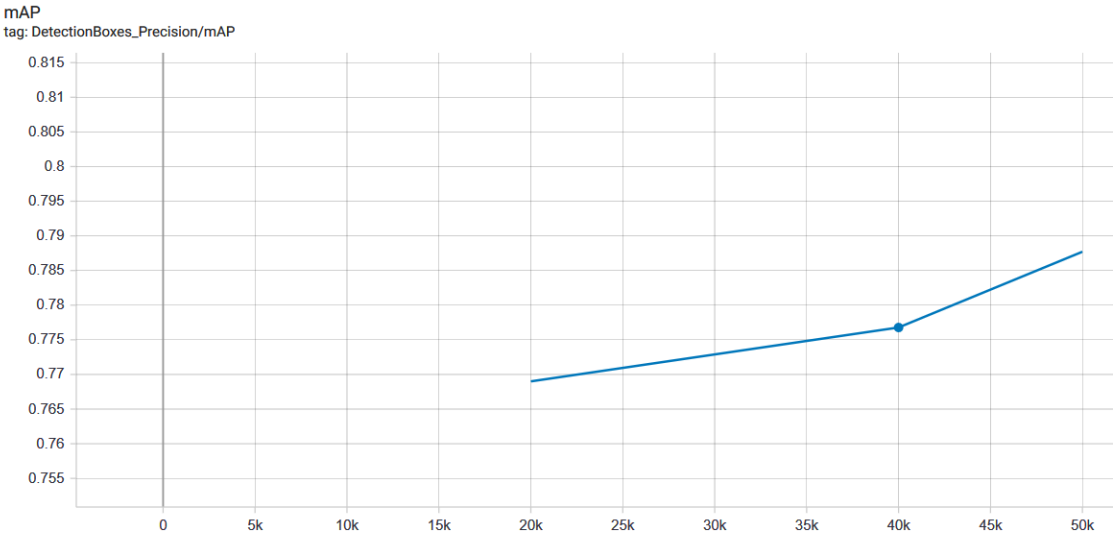
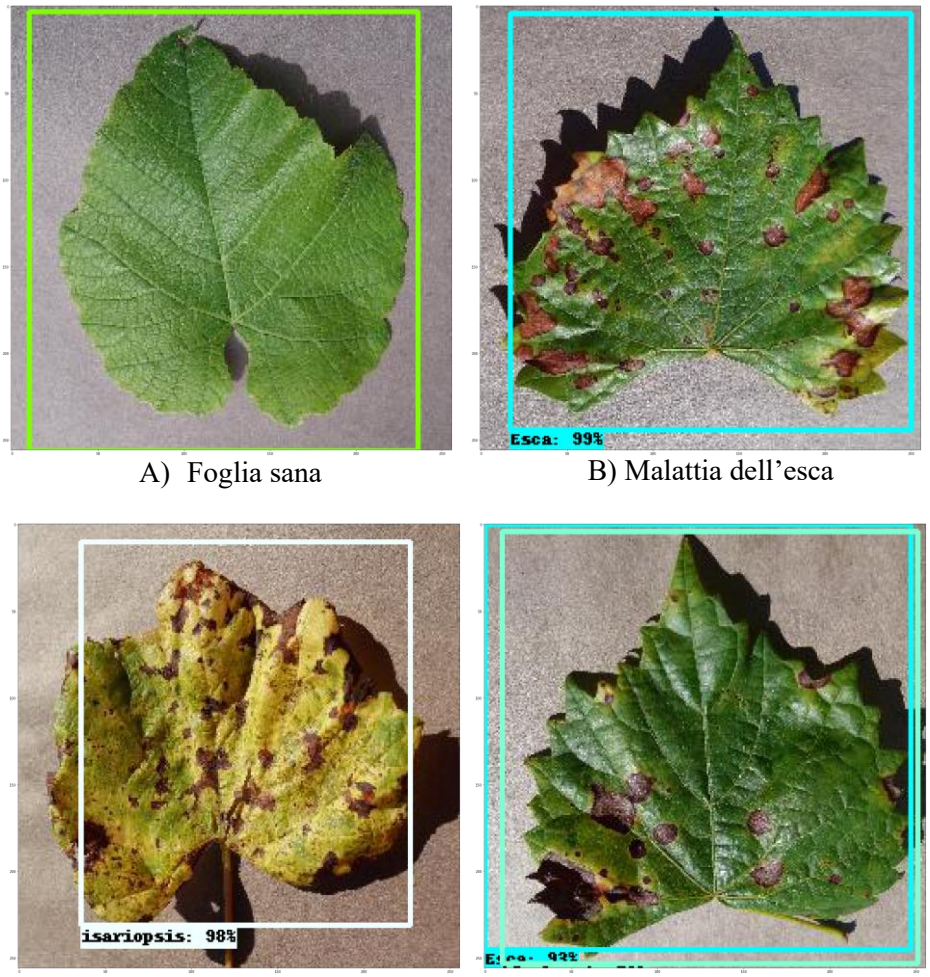


Figura 5.27 mAP in fase di testing.

Per quanto riguarda i valori di mAP sul test set, la Faster R-CNN InceptionV2 ha assunto valori molto simili al caso precedente, evidenziando come entrambe le reti siano riuscite ad eseguire una buona identificazione delle

foto scattate in primo piano delle foglie: allo step 20.000 la mAP ha avuto un valore di 0,76 per poi aumentare a 0,77 allo step 40.000 e infine si è registrato un valore di 0,79 allo step 50.000.



A) Foglia sana B) Malattia dell'esca

C) Isariopsis D) Marciume nero

Figura 5.28 Risultati inferenza per Faster R-CNN InceptionV2.

Osservando il grafico in figura 5.27 e i risultati d'inferenza in figura 5.28, la rete ha una precisione media inferiore alla SSD MobileNetV2 e dimostra anch'essa come non riesce a distinguere le foglie affette da marciume nero da quelle affette da malattia dell'esca evidenziando un carattere più incerto rispetto all'identificazione dei grappoli d'uva. Tuttavia, si può osservare, a occhio nudo, come sia difficile distinguere tra i due tipi di malattie; è un

ottimo risultato, invece, che nessuna foglia malata sia stata scambiata per sana.

5.6.3 Confronto con NCS2

SSD MobileNetV2

benchmark_app	**#Iterazioni**	**Latenza(ms)**	**Throughput(fps)**
CPU	9176	25,14	152,84
NCS2	2636	91,29	43,83

Tabella 5.7 Risultati della benchmark_app per SSD MobileNetV2.

La SSD MobileNetV2 riconferma le prestazioni riportate nella tabella 5.6, nonostante sia stata allenata per molti più step rispetto al caso precedente. Analizziamo i risultati d'inferenza delle stesse foglie con la NCS2:

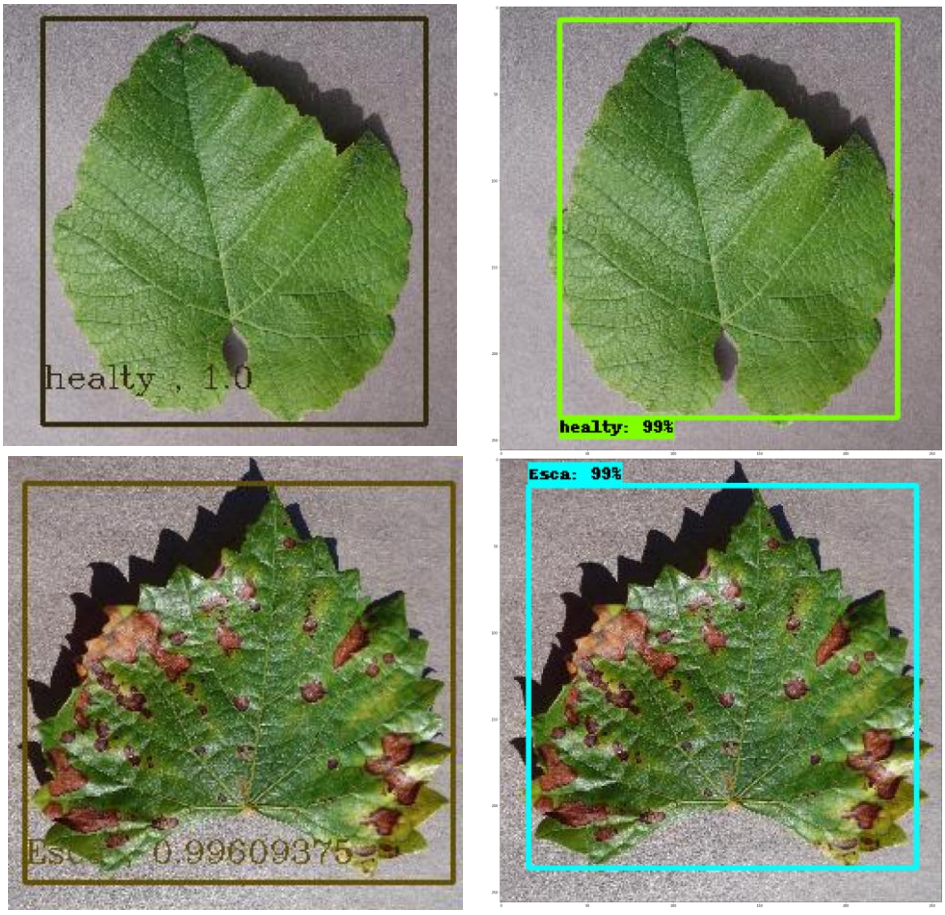


Figura 5.29 A sinistra, risultati dell'inferenza su NCS2, a destra risultati in fase di testing.

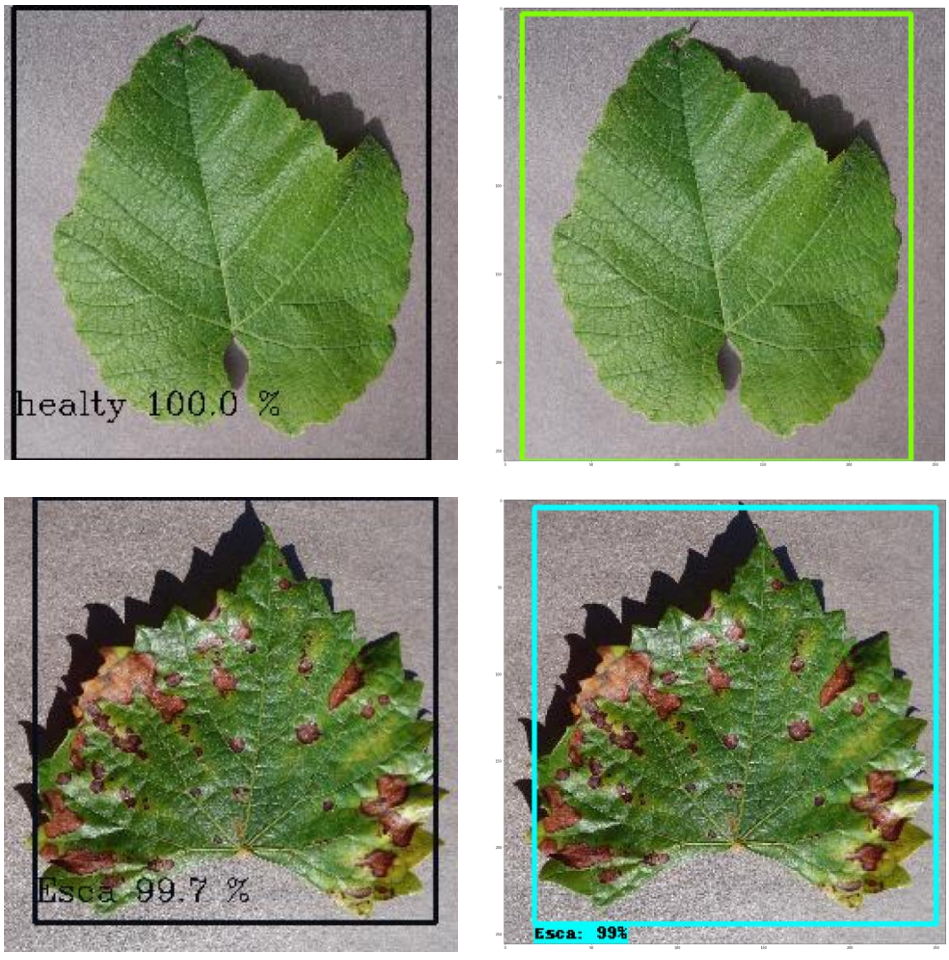
Le previsioni rispecchiano quelle effettuate nella fase di testing del modello; notiamo, tuttavia, che questa volta l'immagine ritraente la foglia affetta da marciume nero, con il 92% di probabilità, è stata classificata con la giusta etichetta, mentre con il 76% con l'etichetta "Esca". Possiamo, dunque, dedurre che l'identificazione delle foglie malate, con questo modello di rete neurale, non è molto affidabile; tuttavia, è un ottimo risultato che venga riconosciuta correttamente la foglia sana.

Faster R-CNN InceptionV2

benchmark_app	**#Iterazioni**	**Latenza(ms)**	**Throughput(fps)**
CPU	164	1537,91	2,62
NCS2	48	5614,22	0,70

Tabella 5.8 Risultati della benchmark_app per Faster R-CNN InceptionV2.

Anche per questa rete, le prestazioni sono pressoché identiche al caso dei grappoli d'uva nella tabella 5.5.



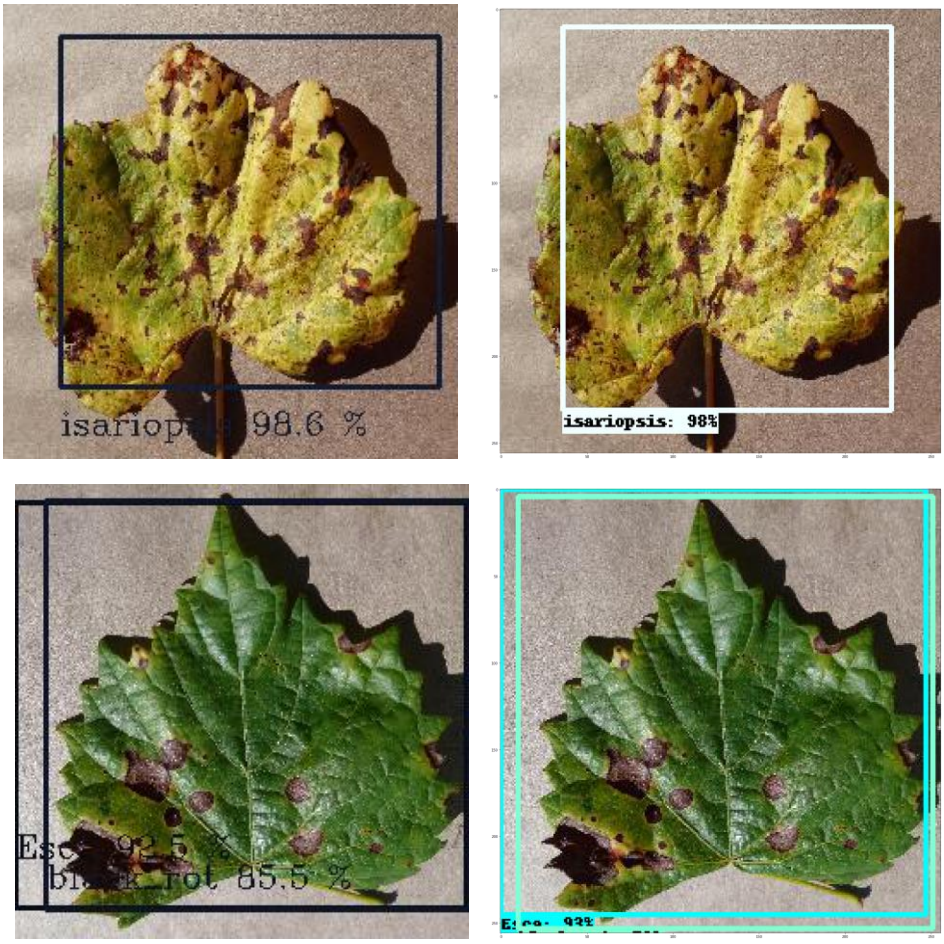


Figura 5.30 A sinistra, risultati inferenza con NCS2, a destra i risultati di inferenza in fase di testing del modello

Anche in questo caso, la Faster R-CNN InceptionV2, utilizzando la NCS2, mostra gli stessi dubbi anticipati in fase di test del modello, riconoscendo la foglia affetta da marciume nero ma continuando a confonderla con una affetta dalla malattia dell'esca; pertanto, considerati i risultati ottenuti e le prestazioni registrate, il modello di rete neurale migliore per il caso dell'identificazioni di foglie di vite malate è stato **SSD MobileNetV2**, poiché garantisce un'identificazione pressoché identica alla Faster R-CNN InceptionV2 ma le sue prestazioni d'inferenza sono nettamente migliori su NCS2, garantendo accuratezza e portabilità.

Capitolo 6

Conclusioni e sviluppi futuri

Il lavoro di questa tesi ha voluto mettere in risalto come il mondo dell'Intelligenza Artificiale e, più in particolare, del Deep Learning sia in continua evoluzione. Basti pensare che la tecnologia dietro la Intel® Neural Compute Stick 2 è nata nel 2017 e questa sua versione, con cui si è avuto modo di lavorare, annunciata nel 2018, ha dimostrato gli enormi passi che si sono avuti in un solo anno di sviluppo, ottenendo un dispositivo otto volte più veloce del suo predecessore. Inoltre, l'avanzamento della tecnica "Inferenza alla periferia" sta a dimostrare come l'Intelligenza Artificiale non è più un argomento legato a giganteschi macchinari dall'elevata capacità di calcolo ma, al contrario, come sia sempre più presente nei dispositivi che ci circondano e come sia utile per ottimizzare il nostro lavoro. Il *focus* sul tema dell'agricoltura di precisione ha voluto mettere in risalto tali caratteristiche; infatti, l'addestramento delle reti neurali artificiali sui grappoli d'uva e sulle foglie di vite malate e il loro successivo utilizzo con il dispositivo vogliono risaltare la duttilità e la portabilità del prodotto e la sua efficacia sul campo, paragonandolo ad una macchina di livello medio-alto come il computer utilizzato.

La ricerca, tuttavia, non è stata esente da limiti, come la lentezza d'esecuzione con una rete neurale più pesante sulla NCS2 e qualche problema nella conversione dei modelli nel formato IR; ciò è dovuto, probabilmente, a una tecnologia ancora "giovane" e ad una documentazione ricca ma ancora da perfezionare nella sua interezza lato software.

Inoltre, le reti neurali per l'identificazione delle foglie di vite malate hanno mostrato una forte incertezza nella distinzione di alcune categorie; dunque, si potrebbe modificare il dataset includendo delle immagini, con relative

annotazioni, di una vigna anziché di sole foglie in primo piano, così da affinarne l'accuratezza. Altresì, si potrebbe sviluppare un applicativo che sfrutti un semplice identificatore di foglie di vite, simile al caso dei grappoli d'uva, che, oltre ad effettuare la localizzazione della foglia nella foto ne esegue una successiva classificazione. Più in particolare, poiché la NCS2, come abbiamo visto, consente il caricamento di molteplici modelli durante l'utilizzo, l'applicativo dopo aver identificato la foglia, dovrebbe ritagliarne i bordi, dare il risultato in pasto a un classificatore e utilizzare il risultato ricostruendo la foto originale con i box identificatori e la classe appena categorizzata.

Infine, si potrebbe proseguire il lavoro, testando l'applicabilità delle reti con la NCS2 in una vigna vera e propria con il supporto di un *Raspberry* installato su un drone o un rover. Infatti, essendo il dispositivo compatibile con *Raspbian OS*, si potrebbe valutarne l'efficacia sul campo in modo che possa essere utilizzato in casi simili con altri tipi di piante e frutti.

Bibliografia

[1] T. Mitchell, *Machine Learning*, McGraw Hill, 1997.

[2] Y. LeCun, Y. Bengio, G. Hinton, *Deep Learning*, Nature, 2015.

[3] F. Pugliese, M. Testi, *Una panoramica introduttiva su Deep Learning e Machine Learning*, URL: https://www.deeplearningitalia.com/una-panoramica-introduttiva-su-deep-learning-e-machine-learning/.

[4] A. Minini, *L'apprendimento supervisionato*, URL: http://www.andreaminini.com/ai/machine-learning/apprendimento-supervisionato.

[5] *Apprendimento non supervisionato*, URL: https://it.wikibooks.org/wiki/Intelligenza_artificiale/Apprendimento_non_supervisionato.

[6] *Reti Neurali*, URL: http://www.intelligenzaartificiale.it/reti-neurali/.

[7] R. Prevete, *Il neurone biologico*, URL: http://www.federica.unina.it/smfn/reti-neurali-e-machine-learning/neurone-biologico/.

[8] K. Kiyoshi, *Arficial Neural Netwoks*, URL: http://wwwold.ece.utep.edu/research/webfuzzy/docs/kk-thesis/kk-thesis-html/node12.html.

[9] D. Beeman, University of Colorado, URL: http://ecee.colorado.edu/~ecen4831/lectures/NNet2.html.

[10] Wikibooks, *Hebbian Learning*, URL: https://en.wikibooks.org/wiki/Artificial_Neural_Networks/Hebbian_Learning .

[11] M. Minsky, S. Papert, *Perceptrons: an introducion to computational geometry,* 1969

[12] P. Medici, *Reti Neurali*, URL: http://www.ce.unipr.it/people/medici/geometry/node107.html.

[13] S. Avinash, *Understanding Activation Functions in Neural Networks*, URL: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0.

[14] Wikipedia*, Funzione SoftMax*, URL: https://it.wikipedia.org/wiki/Funzione_softmax.

[15] D. Rumelhart, G. Hinton, R. Williams*, Learning representations by back-propagating errors*, Nature, 1986.

[16] V. Iuhaniwal, *Forward propagation in neural networks*, URL: https://towardsdatascience.com/forward-propagation-in-neural-networks-simplified-math-and-code-version-bbcfef6f9250.

[17] *Convoltional Neural Networks for Visual Recognition*, URL: http://cs231n.github.io/convolutional-networks/#overview.

[18] A. Krizhevsky, I. Sutskever, G. Hinton, *ImageNet Classification with Deep ConvolutionalNeural Networks*, 2012.

[19] L. Danqing, *A Pratical Guide to ReLU*, URL: https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7.

[20] L. Hulstaert, *Going deep into object detection*, URL: https://towardsdatascience.com/going-deep-into-object-detection-bed442d92b34.

[21] Z. Zhengxia, S. Zhenwei, G. Yuhong, Y. Jieping, *Object Detection in 20 years: A Survey*, ArXiv, 2019.

[22] P. Viola, M. Jones, *"Rapid object detection using abooste cascade of simple features"*. In: *Computer Vision and Pattern Recognition*, 2001. CVPR 2001. *Proceedings of the 2001 IEEE Computer Society Conference on*, vol. 1. IEEE,2001, pp. I–I.

[23] N. Dalal, B. Triggs, *"Histograms of oriented gradientsfor human detection"*. In: *Computer Vision and Pattern Recognition,* 2005. *CVPR 2005. IEEE Computer Society Con-ference on*, vol. 1. IEEE, 2005, pp. 886–89.

[24] P. F. Felzenszwalb, R. B. Girshick, D. McAllester, "*Cascade object detection with deformable part models*". In: *Computer vision and pattern recognition (CVPR), 2010 IEEE conference on.* IEEE, 2010, pp. 2241–2248.

[25] R. Girshick, J. Donahue, T. Darrell, J. Malik, "*Rich feature hierarchies for accurate object detection and semantic segmentation*". In: *Proceedings of the IEEE conference on computer vision and patterm recognition*, 2014, pp. 580–587.

[26] K. E. Van de Sande, J. R. Uijlings, T. Gevers, A. W.Smeulders, "*Segmentation as selective search for object recognition". In: Computer Vision (ICCV), 2011 IEEE Inter-national Conference on.* IEEE, 2011, pp. 1879–1886.

[27] K. He, X. Zhang, S. Ren, J. Sun, "*Spatial pyramid pooling in deep convolutional networks for visual recognition*". In: *European conference on computer vision*. Springer, 2014, pp. 346–361.

[28] R. Girshick, "*Fast r-cnn*". In: *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448

[29] U. Farooq, *From R-CNN to Mask R-CNN,* URL: https://medium.com/@umerfarooq_26378/from-r-cnn-to-mask-r-cnn-d6367b196cfd.

[30] S. Ren, K. He, R. Girshick, and J. Sun, "*Faster r-cnn:Towards real-time object detection with region proposal networks*". In: *Advances in neural information processing systems*, 2015, pp. 91–99.

[31] K. He, G. Gkioxari, P. Dollár, R. Girshick*, "Mask R-CNN". In: Computer Vision (ICCV), 2017 IEEE International Conference on.* IEEE 2017, pp. 2980-2988.

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

[33] J. Redmon, S. Divvala, R. Girshick, A. Farhadi*, "You only look once: Unified, real-time object detection.". In: Proceedings of the IEEE conference on computer vision and pattern recognition,* 2016, pp. 779–788.,

[34] J. Redmon and A. Farhadi, "*Yolo9000: better, faster,stronger*," *arXiv preprint*, 2017.

[35] J, Redmon and A. Farhadi, "*Yolov3: An incremental improvement*". In: *arXiv preprint*, 2018.

[36] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y.Fu, A. C. Berg, "*SSD: Single Shot Multibox Detector*.". In*: European conference on computer vision*. Springer, 2016, pp. 21–37.

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

[38] "*An Introduction to Evaluation Metrics for Object Detection*", URL: https://blog.zenggyu.com/en/post/2018-12-16/an-introduction-to-evaluation-metrics-for-object-detection/, 2018.

[39] K. Wiggers, *Intel's Neural Compute Stick 2 is 8 times faster than its predecessor*, URL: https://venturebeat.com/2018/11/14/intels-neural-compute-stick-2-is-8-times-faster-than-its-predecessor/, 2018.

[40] N. Oh, *Intel Announces Movidius Myriad X VPU, Featuring 'Neural Compute Engine'*, URL: https://www.anandtech.com/show/11771/intel-announces-movidius-myriad-x-vpu, 2017.
[41] P. Alcorn, *Intel Unveils Movidius Myriad X Vision Processing Unit*, URL: https://www.tomshardware.com/news/intel-movidius-vpu-ai-inference,35327.html, 2017.
[42] A. Bai, *Movidius (Intel) annuncia la VPU Myriad X per Deep Learning e AI*, URL: https://www.hwupgrade.it/news/cpu/movidius-intel-annuncia-la-vpu-myriad-x-per-deep-learning-e-ai_70749.html, 2017.
[43] F. Musiari*, FinFET, la tecnologia che estende la vita dei semiconduttori,* URL: https://farelettronica.it/web/finfet-la-tecnologia-estende-la-vita-dei-semiconduttori/, 2016.
[44] WikiChip, *SHAVE v2.0*, URL: https://en.wikichip.org/wiki/movidius/microarchitectures/shave_v2.0.
[45] *Intel Neural Compute Stick 2*, URL: https://software.intel.com/en-us/neural-compute-stick.
[46] *OpenVINO™ Toolkit,* URL: https://docs.openvinotoolkit.org/.
[47] A. Alasdairm, *Benchmarking Edge Computing,* URL: https://medium.com/@aallan/benchmarking-edge-computing-ce3f13942245, 2019.
[48] *Anaconda*, URL: https://www.anaconda.com/.
[49] *TensorFlow*, URL: https://www.tensorflow.org/.
[50] *Tensorflow Object Detection API*, URL: https://github.com/tensorflow/models/tree/master/research/object_detection.
[51] *NVIDIA CUDA toolkit,* URL: https://developer.nvidia.com/cuda-toolkit.
[52] *NVIDIA cuDNN library,* URL: https://developer.nvidia.com/cudnn.
[53] *Tensorflow detection model zoo,* URL: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md, 2019.
[54] *COCO dataset*, URL: http://cocodataset.org/#home.
[55] T. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, P. Dollár*,* "*Microsoft COCO: Common Objects in Context*", 2014.

[56] C. Szegedy, V. Vamhoucke, S. Ioffe, J. Shlens, "*Rethinking the Inception Architecture for Computer Vision*". In: *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[57] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L. Chen, "*MobileNetV2: Inverted Residuals and Linear Bottlenecks*". In: *Computer vision and pattern recognition (CVPR), 2018 IEEE conference on*, IEEE, 2018.

[58] *YOLOv3_TensorFlow*, URL: https://github.com/wizyoung/YOLOv3_TensorFlow, 2019.

[59] *YOLO: Real-Time Object Detection*, URL: https://pjreddie.com/darknet/yolo/.

[60] *LabelImg*, URL: https://github.com/tzutalin/labelImg, 2018.

[61] Google Colaboratory, URL: https://colab.research.google.com/notebooks/intro.ipynb

[62] I. Krasin, T. Duerig, N. AllDrin, A. Veit, S. A. El-Hajia, S. Belongie, D. Cai, Z. Feng, V. Ferrari, V. Gomes, "*OpenImages: A public dataset for large-scale multi-label and multi-class image classification*", 2016. URL: https://storage.googleapis.com/openimages/web/index.html.

[63] *OIDv4_ToolKit: Toolkit to download and visualize single or multiple classes from the huge Open Images v4 dataset*, URL: https://github.com/EscVM/OIDv4_ToolKit, 2019.

[64] *PlantVillage, URL: https://plantvillage.psu.edu/.*

[65] D. P. Hughes, M. Salathe, "*An open access repository of images on plant health to enable the developmetn of mobile disease diagnostics*", ArXiv, 2015.

[66] *PlantVillage-Dataset*, URL: https://github.com/spMohanty/PlantVillage-Dataset, 2018.

Ringraziamenti

Giunti alla conclusione di questa tesi, vorrei riservare questa piccola parte per ringraziare tutte le persone che sono state al mio fianco durante lo sviluppo di tale lavoro e della mia carriera universitaria.

Ringrazio il mio relatore, il Prof. Enrico Masala, per la sua incredibile disponibilità ed i suoi consigli durante tutto il lavoro di tesi, che, senza il cui aiuto, non avrei sicuramente svolto con la serenità con cui digito queste parole.

Ringrazio tutti i miei amici, da quelli della mia città natale a quelli della mia città adottiva, con i quali ho condiviso la maggior parte dei momenti, sia gioiosi che dolorosi; in particolare Francesco, il mio migliore amico, che ha vissuto con me ogni fase di questo percorso sopportandomi e supportandomi nei momenti più difficili.

Ringrazio la mia fidanzata, Francesca, che mi ha aiutato a superare le mie insicurezze ed a calmare le mie inquietudini permettendomi di concludere questo percorso ancora più felice di quando l'ho iniziato.

Vorrei, inoltre, ringraziare anche le persone che sono state al mio fianco in passato e che, chi per un motivo e chi per un altro, adesso non lo sono più, perché è anche grazie a loro se oggi sono felice di essere chi sono.

Infine, un ringraziamento particolare alla mia famiglia, a cui voglio dedicare questo lavoro di tesi, che mi ha sempre sostenuto e ha creduto nelle mie scelte, assicurandosi che fossi sempre felice e che non mi mancasse mai nulla.

A voi tutti, sinceramente, grazie.