POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Valutazione delle prestazioni delle Reti Neurali Convoluzionali su immagini affette da distorsione

Relatore
prof. Enrico Masala

Laureando
Ignazio Barraco

Dicembre 2019

Ringraziamenti

Prima di entrare nel vivo della trattazione, desidero ringraziare vivamente il mio relatore, prof. Enrico Masala, per gli spunti, gli strumenti e per il costante supporto offertomi prima e durante lo svolgimento della tesi e per l'interesse che ha acceso in me verso l'argomento trattato, che sono sicuro mi saranno di grande aiuto nel proseguo della mia vita, lavorativa e non.

Un GRAZIE, un po' più intimo,

A chi mi è stato accanto,

A chi mi ha sostenuto,

A chi ci è voluto essere,

A chi mi ha spinto, spronato,

A chi non ha mai smesso di credere in me,

A chi è tornato a farmi credere in me stesso,

A chi mi è stato di esempio per essere quello che sono oggi.

Sarebbe quasi scontato adesso elencare ciascuno di voi, ma siccome le cose semplici non mi piacciono, lascio a ciascuno di voi che leggerà questa pagina, la possibilità di immaginarsi o meno in una di queste affermazioni oppure chissà solo in qualcuna o in tutte. E semmai, pensiate di esserci, spero in cuor mio di avervi lasciato qualcosa anch'io e di potervela lasciare ancora in futuro. Ma soprattutto un grazie speciale va a voi, Leonardo, Isidoro e Pina, grazie per tutto.

Sommario

Il tema centrale di questa tesi è il Machine Learning e la possibile applicabilità in ambito immagini e video.
Lo scopo finale sarà quello di valutare il comportamento delle reti neurali al variare della qualità intrinseca delle immagini, partendo da quelle originali per finire con immagini che hanno subito distorsioni di vario genere. Il tutto sarà portato avanti testando e sviluppando diverse possibili applicazioni dell'intelligenza artificiale su immagini e video che hanno come denominatore comune la predizione e il riconoscimento dei contenuti. Unendo quanto appena esposto, seppur in maniera generica, verrà fuori il percorso di tesi in oggetto. Per potere avere un quadro della situazione è stato condotto uno studio a priori, su ciò che è stato, su ciò che è e su ciò che potrà essere in futuro, in termini di tecnologie e approcci matematici.

Il lavoro svolto, nello specifico, si articola in 3 macro sezioni: una prima parte in cui è stato fatto un approfondimento prima generico sul Machine Learning, poi un po' più approfondito sulle reti neurali Convolutive (CNN), fornendo anche dei cenni matematici e strutturali; in questa fase preliminare poi è stato analizzato anche lo stato dell'arte dell'intelligenza neurale in ambito immagini e video.
La seconda sezione della ricerca è stata quella inerente allo sviluppo, fase di studio nella quale si è cercato di costruire degli script e dei programmi che permettessero di sfruttare quanto analizzato nella fase precedente.
La fase finale, quella di testing, nucleo centrale della ricerca è quella in cui sono stati studiati e valutati i comportamenti delle reti neurali al variare dell'immagine data in input: quindi immagini distorte in vario modo e con diversi livelli di applicazione della stessa distorsione.
Alla fine della ricerca e dei test sono stati delineati una serie di pattern comportamentali mostrati dalla rete scelta, in diverse circostanze più o meno avverse. Nello specifico si è deciso di analizzare il comportamento di una specifica rete, che desse un buon compromesso tra prestazioni e computazione algoritmica.
Il percorso articolato su 7 capitoli, porterà ad avere delle buone conoscenze sul machine learning in ambito immagini e video, e chissà, magari permettere al lettore di ipotizzare nuovi scenari applicativi o di ricerca.

Indice

Capitolo 1

Introduzione

1.1 Panoramica

L'utilizzo di internet e in particolare dei Social Network ha oramai invaso le nostre vite in maniera del tutto trasparente, inserendosi in maniera centrale nella stragrande maggioranza delle nostre attività giornaliere, siano esse lavorative o ricreative. Tutto ciò ha fatto sì che le problematiche e gli ambiti di ricerca inerenti si siano progressivamente ampliati ed abbiano svariato su diversi fronti.
L'argomento centrale di questa tesi sono le immagini e i video, che caratterizzano internet ma soprattutto i social network. Idea preponderante sarà il riconoscimento dei contenuti all' interno all'interno di questi, al fine di riuscire a predire in maniera sempre più accurata ciò che già naturalmente l'uomo può analizzare.

In questo ambito negli ultimi anni è diventato di fondamentale importanza l'utilizzo di tecniche di Intelligenza Artificiale, quindi di "apprendimento automatico", una disciplina dell'informatica che studia tecniche mirate alla progettazione di sistemi hardware e software utili a far fornire all'elaboratore elettronico prestazioni che comunemente possono apparire solo come di pertinenza dell'intelligenza umana. Oggi l'Intelligenza Artificiale è sfruttata da molti colossi informatici, tecnologici e non: Facebook la sfrutta per i propri algoritmi di tag, Google per la ricerca delle foto, Amazon per indicare consigli sui prodotti e sugli acquisti e così anche altre piattaforme social come Pinterest o Instagram; questi algoritmi riscuotono grande successo nell'identificazione di visi umani oltre che di determinati oggetti in immagini e video oltre che essere molto usate per il riconoscimento di tracce audio e linguaggi naturali.
Un argomento di studio in rapido e costante sviluppo e pertanto una nuova e migliore architettura/algoritmo/tecnica è annunciata da una settimana all'altra o a distanza di mesi, nonostante spesso si tratti di un miglioramento dello stato dell'arte conosciuto fino a quel momento.
L'obiettivo dichiarato e verso cui si sta puntando con decisione in ambito machine learning è pareggiare e in alcuni casi superare i risultati che riesce ad ottenere l'essere umano.

1.2 Scopo della Tesi

Prima di addentraci in maniera dettagliata nella trattazione, è necessaria una doverosa ed importante categorizzazione delle possibili applicazioni tecniche che saranno esposte nel proseguo di questo documento. I task applicativi che saranno trattati sono fondamentalmente 4 e saranno racchiusi in due grandi macro-categorie: la prima categoria riguarderà la classificazione dell'intera immagine e in particolare la labelizzazione del soggetto/oggetto principale, per il quale sarà proposta una percentuale di predizione tra più possibili categorie di oggetti già definite a priori: questo task verrà denominato **image classification** o molto più tradizionalmente **Classificazione**; la seconda categoria porterà ad analizzare la localizzazione dell'oggetto nell'immagine, collocandolo all'interno di un bounding box, proponendo anche in questo caso delle predizioni di riconoscimento per i suddetti oggetti localizzati: questa è la **image classification and localization** o per semplicità d'uso **Localizzazione**. All'interno di questa categoria si può far rientrare un caso più ampio: il possibile riconoscimento di molteplici soggetti/oggetti all'interno di una immagine, con la relativa localizzazione e classificazione per ciascuno: l'**object detection**.
Fondamentalmente addentrandoci negli argomenti, verrà utilizzato per comodità il termine Object Detection o Localizzazione, evitando il riferimento alla presenza di singoli o molteplici oggetti. Sempre in questa categoria si inserisce un sottotask di rilevante importanza che permetterà sia la classificazione sia la localizzazione dell'oggetto ma anche la segmentazione degli oggetti: questi saranno contornati e quindi definiti in maniera più o meno precisa: si parlerà di **istance segmentation**, che può essere visto come lo step successivo di un possibile studio scientifico di ricerca. Su questo task non si è fatto un grosso approfondimento ma sono stati analizzati degli esempi andando a sfruttare un applicativo reperito.
Sarà vista seppur in maniera sommaria anche qualcosa inerente all'object recognition all'interno dei video, sia a livello teorico che a livello pratico testando alcune applicazioni di **object tracking**. Questa può essere vista come terza macrocategoria, ma non avendo sviluppato codice e non avendo fatto una analisi del codice stesso, verrà considerata in maniera marginale.
I due task di classificazione e localizzazione, definibili come i tronconi principali di questa ricerca, saranno quelli su cui verrà valutato il comportamento delle reti: verrà scelta una rete, preferibilmente della stessa famiglia e possibilmente lo stesso modello e si faranno dei test specifici. Verranno date delle immagini in input alla rete, sia in condizioni originali sia in condizioni di distorsione rispetto alla qualità iniziale, verranno analizzati i dati elaborati, e si cercheranno di definire alcuni pattern comportamentali assunti dalla rete in determinate condizioni. Questa fase potrà essere considerata una sorta di fase di testing della robustezza della rete da noi scelta.

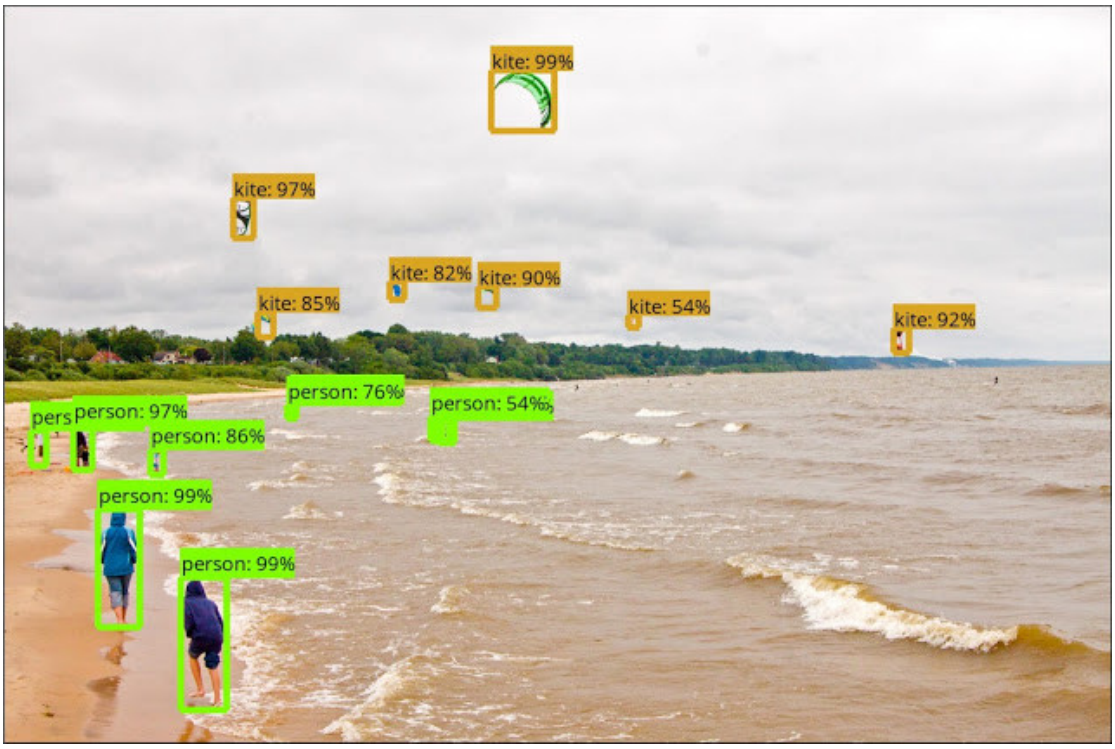


Figura 1.1: In alto a destra un classico esempio di classificazione (MACAW 97,2%) mentre a sinistra di localizzazione e classificazione; in basso un classico esempio di object detection di oggetti multipli.

Capitolo 2

Convolutional Neural Network

2.1 Introduzione al Machine Learning

Nella nostra ricerca si farà uso sia del termine **Machine Learning (ML)** [1] che del termine **Deep Learning [2] (DL)** (e delle relative sigle indicate tra parentesi), consci del fatto che erroneamente si fa spesso confusione, usandoli come sinonimi e come sinonimo del termine Intelligenza Artificiale. Di seguito sarà fatta una panoramica veloce su entrambe al fine di fare chiarezza e comprendere meglio ciò che sarà esposto nei paragrafi successivi.
Machine Learning indica un insieme di metodi con cui si allena l'intelligenza artificiale in modo tale da poter svolgere delle attività non programmate, ma soprattutto da poter apprendere dall'esperienza pregressa come esattamente fa l'intelligenza umana, correggendosi e quindi migliorandosi attraverso gli errori commessi e prendendo decisioni autonome.
Il Deep Learning costituisce l'insieme delle tecniche necessarie alla realizzazione del Machine Learning e quindi può essere considerata una sottoclasse dello stesso. Gli algoritmi di DL simulano il comportamento del cervello umano.
Il ML è un metodo statistico di apprendimento, nel quale ogni istanza, raccolta in un set di dati campionati a priori (immagini nel nostro caso), è descritta da un set di informazioni (features) o attributi, mentre il DL è un metodo statistico per estrarre le informazioni (features) o gli attributi da un da un set di dati mediante l'uso di reti neurali.
Il termine Deep riveste un ruolo chiave: deriva proprio dal fatto che le reti neurali utilizzate sono dotate di svariati livelli nascosti di neuroni, che le rendono per l'appunto profonde e somiglianti a livello strutturale al cervello umano.
L'intelligenza artificiale si lega indissolubilmente con l'ottimizzazione matematica, la quale fornisce metodi, teorie e domini di applicazione. Molti problemi di apprendimento automatico, infatti, sono formulati come problemi di minimizzazione di una certa funzione di perdita (loss function) applicati ad set di esempi (training set). Nel nostro caso, esprime la differenza tra i valori predetti dal modello in fase di allenamento e i valori attesi. "L'obiettivo finale è dunque quello di insegnare alla rete la capacità di predire correttamente i valori attesi su un set di istanze non presenti nel training set (test set) mediante la minimizzazione della loss function". [3]. Sono stati utilizzati termini quali **training set**, **test set**, **loss fuction**: questi sono alcuni degli elementi che saranno dettagliati nei prossimi capitoli.

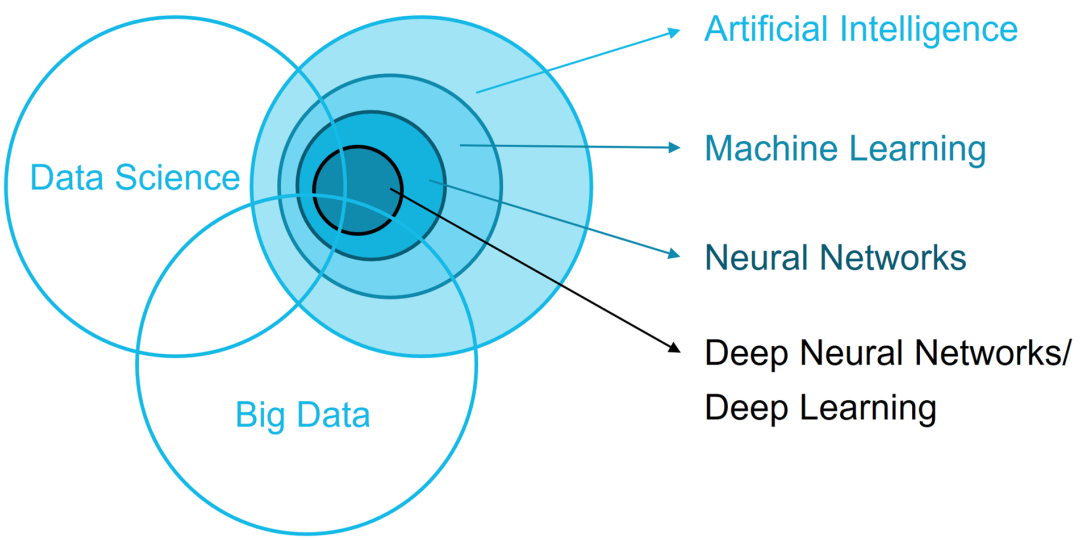


Figura 2.1: Diagramma riassuntivo su Machine learning, Artificial Intelligence, Deep Learning. Fonte: [4].

Quindi saranno analizzati algoritmi in grado di sintetizzare una nuova conoscenza a partire da una base di conoscenza più o meno acquisita in precedenza e si parlerà quindi di algoritmi di apprendimento automatico.
Gli algoritmi di apprendimento automatico sono suddivisibili in due macro-categorie in base al tipo di feedback su cui si basa il sistema di apprendimento artificiale:

- **Gli algoritmi di apprendimento supervisionato** richiedono una base di conoscenza che sia già in seno alla macchina, quindi di istanze di dato annotate similari a quelle che acquisirà; si parla di apprendimento automatico proprio perchè alla macchina vengono passati degli esempi composti da una coppia di dati contenenti il dato originale e il risultato atteso. Il compito della macchina poi sarà quello di trovare una regola che relazioni ciò che già conosce e ciò che arriva in input, in modo che all'arrivo del nuovo dato possa correttamente dare un nuovo risultato in output basandosi sulla regola che ha in memoria.

- **Gli algoritmi di apprendimento non supervisionato** a differenza dei precedenti, non utilizzano dati etichettati o appresi in altre circostanze. L'algoritmo quindi deve creare una nuova regola ricavando i dati necessari dal dato stesso ricevuto in input. Si può anche parlare di **feature learning** (apprendimento di caratteristiche).

- Posto a metà tra il supervisonato e il non-supervisionato, si pone **l'apprendimento parzialmente supervisionato** che si basa su dati misti in cui una parte è dell'apprendimento si fonda su dati già conosciuti e da altri che la macchina deve apprendere istantaneamente.

L'argomento centrale della ricerca può apparire a primo impatto come qualcosa di veramente semplice ma in realtà presuppone la conoscenza di un ampio ventaglio di teoria matematica e innovazioni algoritmiche. Ad esempio, a livello pratico il punto di partenza sarà un'immagine di input utilizzata allo scopo di ottenere in output una immagine "etichettata" nella quale i soggetti rappresentati saranno categorizzati: come già ampiamente sottolineato ciò per noi umani è qualcosa di innato, però quando lavoriamo con una macchina, quale un pc o un dispositivo mobile (con discrete capacità di calcolo), che svolgerà

questa azione la prospettiva cambia totalmente. Questa riceverà in input, semplicemente un array di pixel variabile a seconda della risoluzione e delle dimensioni dell'immagine (per esempio un array 32x32x3 dove 3 è il valore RGB o nel caso di una JPG a colori 480x480 riceverà un array di 480x480x3 elementi) e conoscerà per ciascuno di questi elementi dell'array il valore relativo all'intensità del pixel che varia da 0 a 255.
Lo scenario di osservazione di una immagine tra l'umano e la macchina è quindi diverso ma l'obiettivo prefissato in termini scientifici dai ricercatori è fare sì che la macchina si comporti esattamente come noi riuscendo a differenziare tutte le immagini dicendo quale contiene ad esempio un cane e quale un gatto.

Le Reti Neurali maggiormente usate per raggiungere tale scopo sono le **Reti Neurali Convoluzionali**, conosciute molto più comunemente con la sigla **CNN (Convolutional Neural Network)**. Sarà quindi fondamentale fare una analisi generale sulle Reti Neurali, per poi spostarsi su quelle Convoluzionali. A seguire saranno esposti sia dei concetti matematici e strutturali, che costituiscono la base tecnica da cui partire ma anche dei cenni storici per tracciare l'evoluzione della tematica negli anni.[1]

[1] Le nozioni teoriche sono tratte da diverse fonti: capitoli di libri scientifici trattanti ML [5], articoli su siti specializzati [7] nel settore ma anche da siti di ricerca generica per alcune definizioni [6]

2.2 Neuroni e Reti Neurali

Le reti neurali artificiali (ANN: Artificial Neural Network) [8], come già anticipato, sono modelli matematici che simulano i comportamenti del cervello umano. Il cervello è un calcolatore straordinario: interpreta a velocità elevatissime informazioni fornite dai sensi e impara autonomamente a creare le rappresentazioni interne, che gli permettono di svolgere questa funzionalità. Già a partire dagli anni 90' con i primi approcci si è cercato di riprodurre in maniera artificiale il comportamento del cervello progettando delle sorte di reti (ancora ai tempi embrionali ma oggi molto performanti), costruite tentando prima di individuare le caratteristiche essenziali dei neuroni e delle loro interconnessioni e poi programmando un calcolatore che simulasse tale comportamento.
I neuroni sono internamente costituiti da: **dendriti**, **sinapsi** ed **assoni**. Un neurone raccoglie i segnali provenienti da altre cellule grazie ai dentriti ed emette impulsi elettrici attraverso l'assone, a sua volta diviso in migliaia di diramazioni, alla cui estremità è riconoscibile la sinapsi che trasforma l'attività dell'assone in fenomeni elettrici che inibiscono o eccitano l'attività dei neuroni collegati. Quando un neurone riceve un segnale, invia lungo l'assone un impulso di attività elettrica.
L'apprendimento avviene quando le sinapsi trasmettono i segnali da un neurone all'altro. Traslando tutto ciò al nostro contesto, si nota una similitudine con il comportamento del cervello: le reti artificiali sono composte da unità interconnesse, che corrispondono proprio ai neuroni mentre la funzione della sinapsi è simulata da un peso modificabile associato a ogni connessione. Ciascuna unità trasforma l'insieme dei segnali ricevuti in un unico segnale in uscita, che viene trasmesso alle altre unità.
Tale trasformazione avviene in due fasi: il segnale in ingresso è moltiplicato per il peso della connessione e poi tutti i risultati ottenuti vengono sommati. Nel tempo così il neurone si specializza quindi a riconoscere determinati stimoli e ad apprenderli.

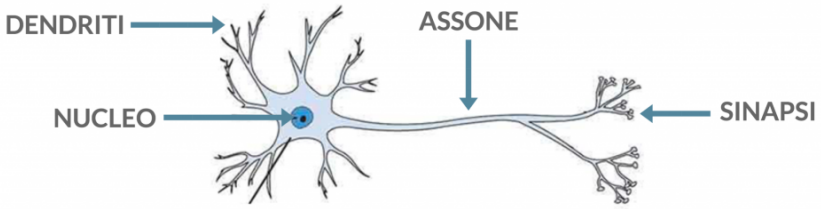


Figura 2.2: Struttura del Neurone. Fonte immagine ed approfondimenti: [9].

Nelle reti Neurali Artificiali sono stati utilizzati due tipi di neuroni artificiali: i **percettroni** e quelli maggiormente usati negli ultimi anni detti **neuroni sigmoidali o di Sigmoid**. Il percettrone prende in input diversi valori binari x1, x2, x3..xn e produce un singolo valore binario in output: per valutare questo valore, bisogna combinare i pesi (wj) corrispondenti ad ogni ingresso (xj). Come detto si tratta di un valore binario e quindi vale 0 o 1, e lo assume a seconda che la somma pesata degli input sia minore o maggiore di un valore di soglia che dipende dal neurone.

Il **percettrone** (fig 2.3) nell'ottica della nostra ricerca può essere assimilato come un modello matematico di classificazione che valuta i valori ricevuti in input: un singolo percettrone non è abbastanza per costruire una vera e propria rete complessa che prenda decisioni complesse, ripetute e in cascata, e magari decisioni sempre più granulari, per cui sarà necessario costruire degli strati detti layer costituiti da diversi percettroni.

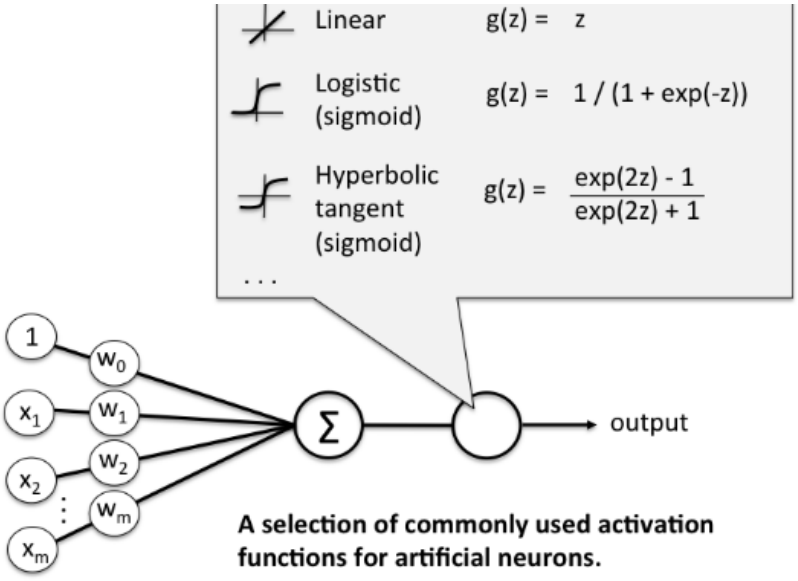


Figura 2.3: Percettrone e possibili funzioni di attivazione. Fonte ed approfondimento: [10].

$$y = \begin{cases} 0 & \text{se} \sum_i w_j x_j \leq soglia \\ 1 & \text{se} \sum_i w_j x_j > soglia \end{cases} \tag{2.1}$$

La funzione 2.1 può essere riscritta ottenendo la 2.2 e definendo come **scostamento** il valore b (matematicamente equivalente a "-soglia"); attraverso questa semplificazione possiamo definire la precedente funzione come la misura di quanto sia facile attivare il neurone: ogni variazione nei pesi o nello scostamento può fare variare l'uscita, e questo può essere sfruttato per far sì che la rete si comporti in modo diverso al variare della situazione.

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

Questo è il punto cruciale alla base delle reti neurali nella fase di apprendimento: si varia un parametro, che sia il peso o lo scostamento della rete, per avvicinarsi a modelli di predizione sempre più corretti e precisi. L'insieme dei pesi e del valore di scostamento costituiscono il **bias** e rappresentano proprio l'informazione che il neurone apprende in fase di addestramento e che conserva per le successive riutilizzazioni.
Il percettrone, pur avvicinandosi a questo concetto ideale non è il migliore dei neuroni artificiali, visto che è un classificatore binario che fornisce quindi in output 0 e 1: questo dettaglio è limitante poichè non ci permetterà di osservare un cambio significativo nel modello ma solo una variazione alla modifica dei bias. Il neurone artificiale che assimila meglio tale comportamento è invece il **Neurone Sigmoidale** rappresentato nel medesimo modo del percettrone e che può avere quindi fino a n input (x1, x2..xn) con i relativi pesi (w1, w2..wn) e che dipende dallo scostamento b; ha un uscita y, che può essere applicata nuovamente come ingresso per altri neuroni sulla rete: la differenza sostanziale con il percettrone sta nel fatto che il valore che possono assumere gli ingressi e le uscite sono valori che oscillano tra 0 e 1.
Questo permetterà di modellare sistemi realistici.
La funzione di uscita 2.3 è di seguito definita e dipende dalla funzione sigmoidale. 2.4:

$$y = \sigma(w \cdot x + b) \tag{2.3}$$

$$\sigma(z) \equiv \frac{1}{1 + e^{-}z} \tag{2.4}$$

Dal punto di vista matematico mentre la funzione del percettrone risulta essere una funzione a scalino quella del neurone sigmoidale è una curva molto lenta che la rende adatta a fornire in uscita cambiamenti sempre piccoli al variare dei parametri.

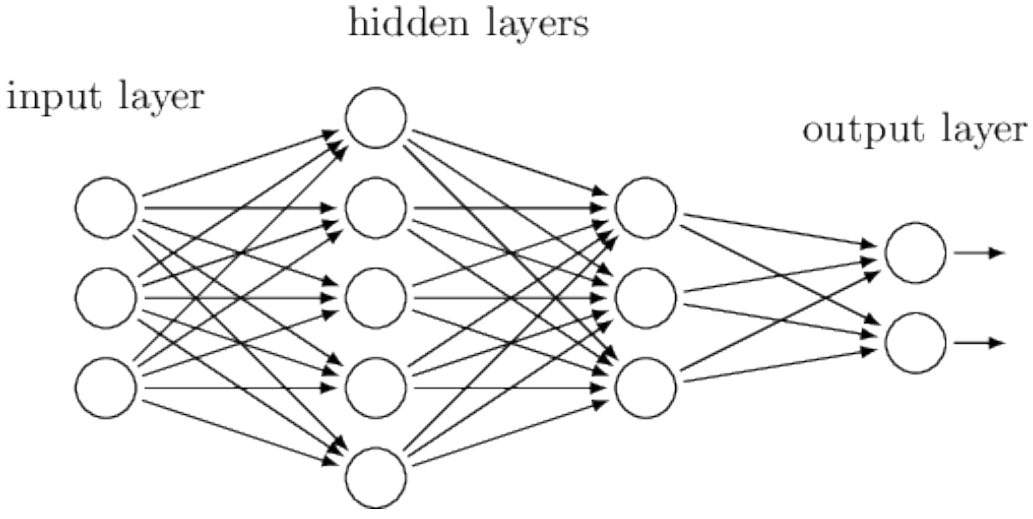


Figura 2.4: Struttura a livelli di una rete neurale. Fonte immagine: [11].

Una rete neurale quindi risulta essere costituita da uno strato di input e da uno di output (figura 2.4), che in figura sono i più esterni mentre gli strati interni dipendono dal tipo di rete implementata e determinano la profondità della rete; il numero di input dipende dallo scopo e dalle scelte implementative mentre lo strato di output è variabile in base al numero di valori in uscita del modello, e la presenza di più neuroni di uscita può rendere più o meno preciso il risultato.
Un output, in questa struttura, può essere inviato come input allo strato successivo creando delle **reti feed-forward** a patto che non vi siano dei cicli, quindi l'informazione viene sempre inviata in avanti; se ipoteticamente creassimo dei cicli potremmo parlare di **Reti Neurali Ricorrenti (RNN-Recurrent Neural Network)**, che si avvicinano più al comportamento umano ma che sono meno utilizzate in termini di applicazione, computazione e risultati e non saranno viste in questa ricerca.

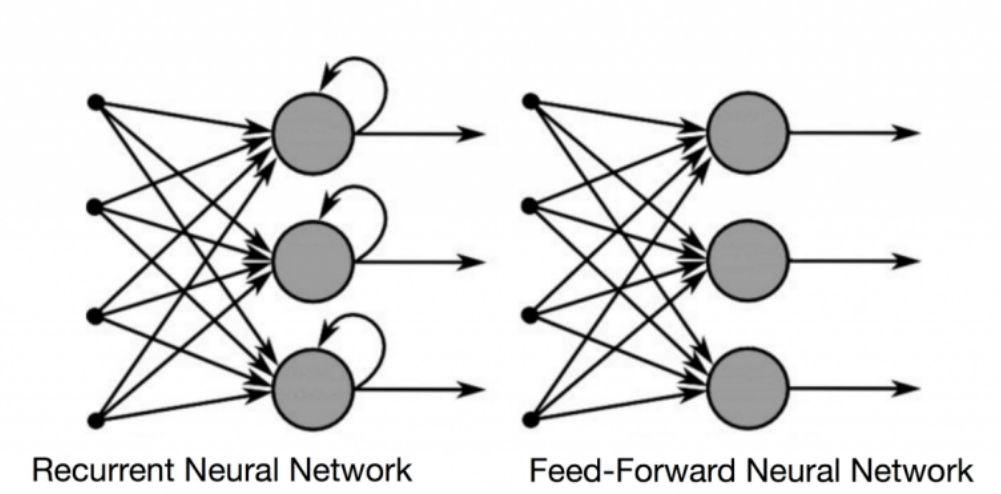


Figura 2.5: RNN vs Feed Forward. Fonte immagine ed approfondimenti: [12].

2.3 La Convoluzione

Proposta una panoramica molto generale sulle Reti Neurali si può adesso scendere più nel dettaglio analizzando quelle che sono le reti di nostro interesse: le **Reti Neurali Convoluzionali**.

2.3.1 Cenni Matematici

Convolutional Neural Network è un nome che nasce dall'operazione matematica sfruttata, per l'appunto la Convoluzione, che è un tipo particolare di operazione lineare.
Le CNN sono progettate per riconoscere dei pattern visivi in modo diretto e non richiedono molto preprocessing o comunque ne richiedono una quantità molto limitata; si ispirano al modello della corteccia visiva animale: i singoli neuroni in questa parte del cervello rispondo solamente a stimoli relativi ad una zona ristretta del campo di osservazione detto campo recettivo.
Una prima premessa, doverosa, va fatta in merito alle formule viste in precedenza relativamente a percettroni (2.1) (2.2) e neuroni sigmoidali (2.3) (2.4): le variabili in gioco cioè gli input possono essere oltre che in forma vettoriale anche in forma scalare o matriciale.
La rete Convoluzionale è quindi una struttura neurale che sfrutta la convoluzione al posto di una semplice moltiplicazione di una generica matrice di per i valori di uno dei layer. Però, la convoluzione in questo contesto, non è propriamente l'operazione matematica tradizionale usata in altri campi dell'ingegneria.

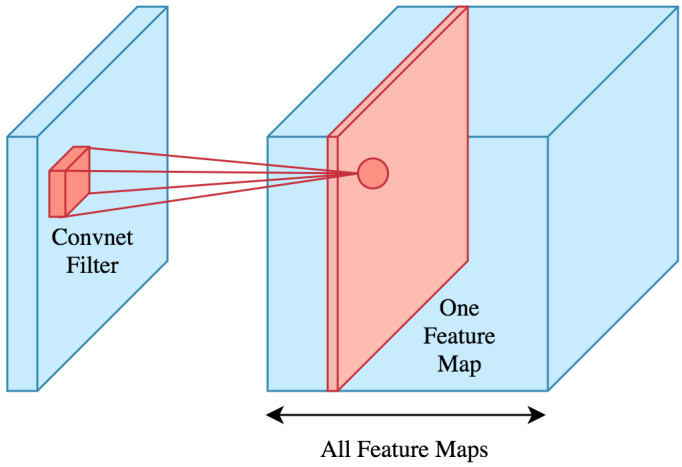


Figura 2.6: Idea alla base della convoluzione. Fonte ed approfondimenti: [13].

La Convoluzione, infatti, riprende le funzioni viste in precedenza (2.1) (2.2) (2.3) (2.4) che saranno sfruttate come punto di partenza e che dipendono da due funzioni con argomenti reali: una funzione pesata **(a) detta kernel** ed una **funzione x di input**. Dall'applicazione di queste verrà fuori una funzione risultante s come nell'equazione (2.5). Il simbolo della convoluzione è un asterisco e il risultato ottenuto sarà una feature map.

$$s(t) = (x * w)(t) \tag{2.5}$$

Lavorando con dati "informatizzati" o per meglio dire "digitali", il tempo risulta essere discretizzato e dalla precedente formula si otterrà una formula più appropriata (2.6):

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a) \tag{2.6}$$

L'input è un array multidimensionale di dati mentre il kernel è un array multidimensionale di parametri che dipendono dall'algoritmo di learning scelto.
Entrambi questi array spesso sono chiamati **Tensori**. Riprendendo la formula (2.6), la sommatoria infinita potrà essere vista come una sommatoria su un numero finito di elementi di un array, su cui applicare la convoluzione in uno o più assi del tempo: ad esempio se ricevessimo una immagine I in 2D come input dovremmo usare un kernel in due dimensioni.

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.7}$$

Considerando anche la commutatività della convoluzione:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.8}$$

La proprietà commutativa nasce dal fatto che se ipoteticamente aumentassi m, l'index dell'input crescerebbe ma l'index del kernel diminuirebbe. Tutto ciò però è poco interessante dal nostro punto di osservazione dal momento che non è importante per le reti neurali sfruttare la proprietà commutativa.
Le equazioni (2.7) e (2.8) sono indicate in modo improprio come **funzioni di correlazione** e sono quelle applicate nella progettazione di una libreria di learning, al variare del range di m e n. In taluni casi viene spesso usata una funzione detta di Cross-Correlazione (2.9), che corrisponde alla funzione di correlazione su cui non è stata applicata l'inversione del kernel (flipping del kernel).

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{2.9}$$

Come visto, le reti neurali tradizionali ricevono in input un singolo vettore e lo trasformano attraverso una serie di strati nascosti, nel quale ogni neurone è connesso ad ogni singolo neurone sia dello strato precedente che di quello successivo. Nel caso l'input sia costituito da immagini di dimensioni ridotte, ad esempio 32×32×3 (32 altezza, 32 larghezza, 3 canali di colore), un singolo neurone connesso nella suddetta maniera comporterebbe un numero totale di $32 \times 32 \times 3 = 3072$ pesi, quindi un numero considerevole di parametri di partenza; le dimensioni però, in realtà, non sono queste ma ben più elevate: appena 256 pixel fanno crescere il carico a $256 \times 256 \times 3 = 196.608$ pesi per singolo neurone, ovvero quasi 2 milioni di parametri per una semplice rete con un singolo livello composto da dieci neuroni. L'architettura appena riassunta è chiamata **fully-connected** [14] (fig.2.7) e risulta troppo esosa, così le Convolutional Neural Networks si fondano su tre fondamentali proprietà: **l'interazione sparsa (sparse interaction)**, **l'invarianza rispetto a traslazioni (invariant to translation)**, e la **condivisione dei parametri (weight sharing)**. Viene fuori una rete più efficace e allo stesso tempo parsimoniosa in termini di parametri.

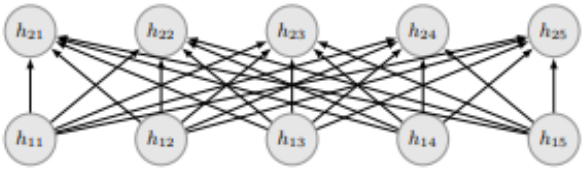


Figura 2.7: Fully Connected Layer. Fonte: [14].

2.3.2 Sparse interaction

L'interazione sparsa [14] (fig. 2.8) permette di ridurre le dimensioni del kernel rispetto all'input: ad esempio processando un' immagine, si dispone di centinaia o migliaia di pixel ma tecnicamente si possono riconoscere solo ridotte quantità di feature significative. Questo sta a significare che si può ridurre il numero di parametri da salvare, riducendo la richiesta di risorse per il modello e aumentando la sua efficienza.
In pratica con m input e n output allora la matrice moltiplicativa sarà di m x n con complessità O(m x n); limitando il numero di connessioni che l'output può avere verso k si otterrà una complessità O(k x n) e un miglioramento delle prestazioni.

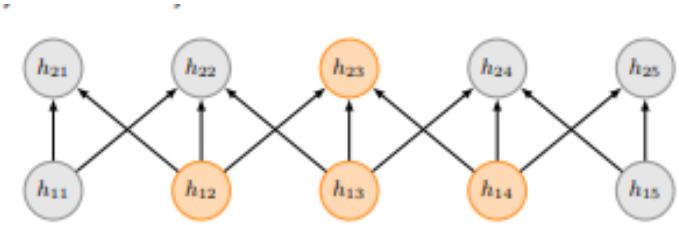


Figura 2.8: Sparse Interaction. Fonte: [14].

2.3.3 Parameter sharing (Weight sharing)

L'idea è quella di condividere parametri tra più funzioni in un modello. Nelle reti neurali tradizionali ogni elemento della matrice pesata è pesato solo una volta per ottenere l'output del singolo layer e viene moltiplicato per l'input senza mai essere modificato nel tempo, risultando riutilizzabile a posteriori.
Come sinonimo di parameter sharing potremmo parlare di **Tied Weights**, poichè il valore dei pesi applicati ad un input è legato al valore dei pesi applicati dovunque.
In una rete Convoluzionale oltretutto ogni membro del kernel è usato da tutti gli input e quindi la condivisione dei parametri offre il vantaggio di imparare un unico set di parametri piuttosto che un set di parametri distinto per ogni locazione.

2.3.4 Equivariant representation (invarianza alle traslazioni)

È una proprietà che il layer acquisisce grazie alla condivisione dei parametri e nello specifico una funzione si dice **equivariante** quando al variare dell'input vi è una uguale variazione dell'output. Proprio come si erano notato per i neuroni artificiali per quanto riguarda la loro applicabilità nella costruzione di reti artificiali di apprendimento. Nel caso specifico della convoluzione se avessi una funzione g() che trasli l'input, allora la funzione di convoluzione risulterebbe equivariante rispetto alla funzione g(). Dando in input una serie di dati si creerebbe una sorta di finestra temporale in cui andare ad individuare quali siano le feature dell'input. Nel nostro caso, invece, passando in input una immagine 2D la Convoluzione creerà una mappa di feature (quindi di elementi salienti) dell'input e nel caso venisse traslato l'input o venisse mosso un oggetto nell'immagine di input, nell'output ci sarebbe un movimento proporzionato.
La convoluzione però non è naturalmente equivariante a tutte le trasformazioni ma solamente a cambi in scala o a rotazioni delle immagini e questo sarà un elemento da valutare nelle prestazioni finali di una rete.

2.4 Struttura di una CNN

Le reti neurali sono costituite da svariati livelli alternati tra loro, che principalmente saranno di tipo convolutivo o di sottocampionamento (**pooling layer**); oltre agli appena citati, di comune utilizzo risultano essere i **ReLu** layer e i **layer fully connected**, livelli fortemente connessi che sono posti alla fine della struttura della rete. Questi layer tutti insieme costituiscono il blocco base solitamente presente in ogni CNN e sono posti tra di loro in cascata.

- **Input Layer**
  Il primo layer è quello che riceve l'immagine e la ridimensiona prima di passarla ai layer successivi.

- **Convolutional Layer**
  Lo scopo dello strato di Convoluzione è quello di estrarre le features: sono le caratteristiche significative delle immagini, usate per poi calcolare i match tra i punti caratteristici durante il testing: è il livello principale della rete.
  Si cercano di individuare dei pattern, come ad esempio curve, angoli, circonferenze o quadrati raffigurati in un'immagine con elevata precisione.
  I livelli convolutivi possono essere molteplici ma questo dipende dall'architettura di rete: maggiore è il loro numero, maggiore è la complessità delle caratteristiche che riescono ad individuare.
  La matrice, che a livello matematico è detta Kernel, è il filtro (di cui viene indicata solitamente la dimensione, ad esempio 3x3) che verrà convoluto con l'input e permetterà di ottenere una **Activation Map** detta **Feature Map**. È chiaro che cambiando il filtro con la stessa immagine di input otterrei delle mappe risultanti diverse.
  Facendo un riassunto prettamente pratico ma anche semplicistico, si può dire che una CNN apprende i valori di questi filtri durante il training e si evince che quanti più filtri si useranno quante più features si estrarranno.
  La feature Map ottenuta dipenderà dalla profondità (Depth) cioè il numero di filtri usati, dallo Stride che è il numero di pixel di cui ci sposta con la nostra matrice di filtro sulla matrice di input e dallo Zero-Padding cioè il numero di zeri di padding usati per completare la matrice di input lungo i bordi (wide convolution).

- **Rectified Linear Unit Layer (RELU)**
  Questa è una operazione aggiuntiva, usata subito dopo la Convoluzione ed effettuata pixel per pixel. Con la Relu viene introdotta la **non linearità** nella rete, dal momento che ogni dato reale che sarà usato sarà non-lineare, al contrario della Convoluzione.
  Questo layer sostituisce ogni numero negativo del pooling layer con il valore zero permettendo alla CNN di restare matematicamente stabile e di mantenere i valori ottenuti bloccati vicino allo zero o comunque facendoli tendere verso l'infinito. Viene spesso utilizzata per il training proprio perché il calcolo della sua derivata è veloce ed il valore che questa assume è zero solo se l'ingresso è minore di zero.

- **Pooling Layer**
  I set di features ottenuti sono passati a questo strato che prende immagini di grandi dimensioni e le contrae preservando le informazioni fondamentali. Il pooling spaziale può essere di diversi tipi, tra cui il più famoso ed usato è il **max-pooling**, che riprende il concetto del vicino spaziale prendendo il più grande elemento della mappa dentro la finestra scelta. Permette di identificare se la caratteristica di studio è presente nel livello precedente.
  Questo livello, inoltre, rende l'input delle features più piccolo e gestibile riducendo il numero di parametri sulla rete e di conseguenza ottimizzando la computazione; rende inoltre la rete invariante alle piccole trasformazioni quali distorsione o traslazione rispetto all'immagine iniziale.

- **Fully Connected Layer**
  È solitamente posto alla fine della rete e si occupa di prendere le immagini filtrate ad alto livello e tradurre in categorie ciò che ha analizzato, quindi il suo scopo è quello di usare le features per classificare le immagini. Ogni classe rappresenta una possibile risposta finale che il computer darà.
  Si tratta di un **MultiLayer Perceptron** che usa una funzione di attivazione **Softmax** ed è proprio questa che permette che la somma delle probabilità di un layer di questo tipo sia 1: nello specifico prende un vettore di valori reali arbitrari (punteggi) e offre in uscita un vettore con valori tra 0 e 1 che sommati danno esattamente 1. Il termine fully connected implica che ogni neurone del layer precedente è collegato ai neuroni del layer successivo.
  Chiaramente questa è una overview basilare della struttura mentre nel capitolo storico-tecnico saranno proposte delle varianti struttuali ottenute aggiungendo layer o modificandone l'ordine allo scopo di ottenere prestazioni migliori e/o soluzioni che richiedano meno computazione.

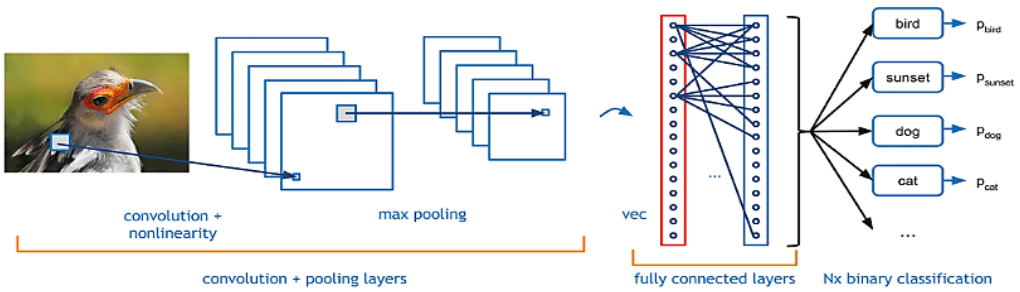


Figura 2.9: Tipico esempio di struttura Cnn per classificazione [15].

2.5 Algoritmi di Pooling

Nell'architettura di una CNN è pratica comune inserire fra due o più layer convolutivi uno strato di Pooling, la cui funzione è quella di ridurre progressivamente la dimensione spaziale degli input (larghezza e altezza), in modo da diminuire il numero di parametri e il carico computazionale. Di seguito saranno evidenziati alcuni tra gli algoritmi di pooling utilizzati nella costruzione di reti convolutive.

2.5.1 Global Average Pooling

Il **GAP** negli ultimi anni è stato parecchio usato per ridurre l'overfitting dovuto alla riduzione del numero dei parametri all'interno della rete o anche per ridurre le dimensioni dei tensori 3-D.
In particolare, riesce a permettere una riduzione estrema della dimensionalità riducendo un tensore dalla dimensione H x W x D alla dimensione 1 x 1 x D. Il layer GAP riduce ogni feature map in un singolo numero che riassume i valori ottenuti, utilizzando una media dei valori del layer precedenti e della feature map.

2.5.2 Max-Pooling

Il **Max-Pooling**, citato già in precedenza, è stato molto spesso usato nella costruzione di reti neurali; in questo livello viene calcolato il valore massimo o comunque il più grande per ciascuna feature map. Il risultato viene poi sottocampionato o raccolto insieme agli altri valori in modo da creare una rappresentazione che comprenda i valori più presenti: da sottolineare che non si tratta però di una media matematica.

2.5.3 Spatial pyramid Pooling

Lo **Spatial Pyramid Pooling (SPP)** [16] è un livello di pooling, che spesso viene implementato in livelli interni multipli, e a differenza dei due precedenti ha avuto grande applicazione nelle reti di Object Detection, quindi per utilizzare reti **"RPN (Region Proposal Network)"** nelle quali vengono proposte delle regioni spaziali in cui possibilmente è possibile trovare oggetti.
Solitamente le reti neurali, attraverso alcune operazioni aggiuntive rimuovono la limitazione dell'utilizzo di immagini con dimensione fissata pur sapendo che questo può ridurre alla lunga la loro precisione e accuratezza: con questo tipo di pooling viene eliminata questa condizione limitante. L'operazione nello specifico crea una feature map da una intera immagine, per poi fare un pooling delle feature in regioni arbitrarie in modo da generare delle rappresentazioni di dimensioni fissate, utili ad allenare il detector di oggetti sulle immagini. In altre parole, viene fatta una aggregazione di informazioni tra il livello convolutivo e quello successivo che può essere un Fully Connected o un livello che si occupa di Object Detection.

2.5.4 ROI Pooling

Region-of-Interest (RoI) Pooling [54] [96] è un tipo di pooling layer che completa il max-pooling di un input in forma non stabilita e produce una piccola feature map di dimensione stabilita cioè 7x7. Questa dimensione è un iperparametro della rete. Ciò ha permesso una velocizzazione della fase di allenamento, rendendo le architetture che ne fanno uso tra le più veloci nel campo dell'Object Detection.
Molto più dettagliatamente è utilizzata per produrre in un singolo passaggio una singola feature map da tutte quelle proposte dalle reti RPN (Region Proposal). In qualche modo anche questa risolve il problema di avere delle immagini di dimensioni fisse in input.
Nella costruzione del layer vanno appositamente settati 3 parametri che non saranno dettagliati in questa trattazione:
pooled_width, pooled_height, spatial_scale.

2.6 Progettazione di una Convolutional Neural Network

Nella fase di progettazione di una CNN e comunque anche in quella di una rete neurale ci sono determinati passi da eseguire più o meno facoltativi.

2.6.1 Il modello

Il modello è la tipologia di rete neurale che sarà scelta per l'attività programmata: ovviamente sarà una CNN appartenente a una delle diverse famiglie di modelli come le VGG-NET O le ResNet, delle quali sono conosciute diverse versioni, differenti tra loro e più o meno ottimizzate.

Scelta del modello

Come primo passaggio quindi bisognerà andare a valutare quale tipo di rete andare ad usare, più o meno in base alla potenza di calcolo disponibile e al tipo di analisi o task da svolgere.
Modelli troppo semplici possono portare a problemi di underfitting mentre modelli esagerati possono portare ad overfitting, quindi questa è una scelta di fondamentale importanza in quanto caratterizzerà ogni scelta successiva e ogni risultato e predizione. Ovviamente l'utente può pensare di costruire un nuovo modello di sana pianta, componendolo con i layer visti in precedenza ma solitamente si parte da dei modelli già conosciuti che costituiscono lo stato dell'arte attuale.

2.6.2 Il Dataset

Il dataset come si può facilmente evincere dal nome è un set di dati, nel nostro caso di immagini, usate per testare ed allenare l'architettura di rete. Esistono tre tipi di dataset: **Training-Set**, **Validation-Set** e **Test-Set**.

Scelta del dataset

Si può scegliere di utilizzare un dataset preesistente (successivamente sarà fatta una carrellata dei più famosi dello stato dell'arte), di crearne uno partendo da uno dei noti magari riducendone le dimensioni o in alternativa di crearne uno totalmente personalizzato con delle immagini che devono essere ridimensionate in modo adeguato, seguendo delle regole dettate dal modello di rete scelto.
In generale servirà creare un Training-Set che sarà usato per permettere al modello di rete, scelto nel primo passo, di imparare; un Validation Set per fare delle valutazioni sul modello, anche al fine di migliorare i parametri già appresi ed affinare le prestazioni della rete ed infine un Test-Set per provare a fare delle prove pratiche sul modello già finito e già allenato. Spesso parte del validation set viene usato anche come Test-Set.

2.6.3 Modifica del modello di rete

Dopo aver deciso per un determinato modello, potremmo prendere in considerazione la possibilità di apportare delle modifiche: sarà necessario fare numerose sperimentazioni in modo da non intraprendere delle scelte azzardate o totalmente errate.
Si possono progettare delle modifiche a livello strutturale o si possono variare gli iperparametri, la cui scelta è una operazione matematica oltre che intuitiva, che influenza le prestazioni finali della rete: gli iperparametri infatti caratterizzano la Convoluzione e l'operazione di train dell'algoritmo. Oltre agli appena citati, vi sono altri elementi di scelta arbitraria di grosso impatto: la dimensione dell'input visto che le immagini in ingresso alla rete devono essere ridimensionate prima di applicare la prima Convoluzione, il numero di layer di Convoluzione, il posizionamento del pooling visto che si può decidere di ridurre il numero di features mediante pooling dopo ogni layer o si può anche decidere alternativamente di non fare pooling. Se però si decidesse di fare pooling diverrebbe fondamentale proprio la scelta appropriata della funzione, tra le quali la Max-Pooling appare essere la più performante oppure si potrebbe decidere di usare diverse funzioni di pooling al variare del livello.
Un'altra scelta consistente è quella inerente alla dimensione dei filtri, poiché ogni layer di Convoluzione può usare dei filtri con differenti tipi di "receptive field". Altre possibili variabili di scelta affidate all'utente possono essere la scelta numerica dei Layer Fully-Connected, la scelta della Loss Function o della funzione di non linearità atta all'attivazione dei neurali (la quale può essere differente per ogni layer).
Tutte queste scelte, che possono modificare il blocco base di una rete, possono apportare delle migliorie ma potrebbero essere deleterie per le prestazioni, qualora qualuna delle scelte implementative fosse totalmente errata o priva di fondamento.

2.6.4 Training Della rete

La fase pratica di apprendimento (learning) è la seguente: vengono prima di tutto inizializzati i filtri, i parametri ed i pesi, e vengono settate tutte le varie opzioni di training in accordo anche con le specifiche della GPU disponibile; fatto questo sarà possibile iniziare ad allenare la rete con il training set scelto. È una operazione molto dispendiosa a livello computazionale. Il Training però può essere facoltativo in quanto le reti neurali ci propongono diversi scenari di utilizzo: infatti in alcune circostanze ci si può imbattere in passaggi diversi in cui non vi è un allenamento della rete ma viene svolta solo una operazione di **Transfer Learning** oppure viene utilizzata una rete pre-allenata della rete.

Transfer Learning

È pratica più comune sfruttare una rete già addestrata e utilizzarla per un secondo o successivo task. In particolare, vengono presi i pesi dall'addestramento di qualcun altro e trasferiti per testare una rete o per riallenare un modello simile: la rete può partire quindi con dei pesi già pre-allenati.
A sua volta vi sono due sottoutilizzi: **Feature extractor** nel quale si rimuove l'ultimo layer (fully-connected) e si usa il resto della rete come feature extractor su un nuovo set di dati su cui lavorare oppure il **Fine-tuning** nel quale vengono ottimizzati i pesi della rete preaddestrata.
In questo modo le features generiche della rete diventano piano piano sempre più specifiche per la nuova attività.

Pretrained models

Le nuove reti convoluzionali richiedono per l'allenamento 2-3 settimane pur sfruttando multiple GPU, così spesso si cerca di sfruttare l'addestramento e il fine-tuning fatto e condiviso da altri utenti e si va ad usare la rete già pronta per un nuovo task. Nell'applicazione pratica della ricerca in oggetto ci si soffermerà su questa attività. Una rete pre-allenata nello specifico è una rete che è stata allenata sulla stessa tipologia di dati di input e magari sulla stessa applicazione. Quindi diventa fondamentale reperire oltre che il dataset anche delle reti pre-allenate coi relativi parametri, in modo da poterle sfruttare in maniera diretta.

2.7 La fase di addestramento di una rete Neurale

Nella precedente sezione è stato approfondito il concetto del training della rete (attività che non sarà di nostro interesse pratico). Nei capitoli successivi ci si imbatterà in altri concetti di difficile comprensione, che senza alcun tipo di background potrebbero sembrare qualcosa di astratto: di seguito allora verrà fatta una descrizione ad alto livello di queste nozioni, in modo da non restare scoperti su elementi utili alla comprensione di ciò che sarà visto.
Gli algoritmi di apprendimento supervisionato, categoria in cui sono inquadrati gli algoritmi di classificazione che saranno studiati, lavorano in due fasi: il **training (Addestramento)** che è stato già accennato e il **testing** in cui ciò che è stato appreso viene sfruttato per classificare nuovi elementi del set di dati.
Nella fase di training bisogna stimare i pesi, con delle tecniche di ottimizzazione matematica come quella della **discesa del gradiente (gradient descent)** mediante **back propagation** [17] basata su due fasi cicliche: propagazione e aggiornamento dei pesi. Nella fase di **propagazione (forward step)** gli elementi di input attraversano l'intera rete per poi recuperare, dopo la propagazione, gli output ottenuti in precedenza.
Viene calcolato l'errore di predizione attraverso la loss function, con cui calcolare il gradiente che verrà poi propagato all'indietro nella rete.
La seconda fase di aggiornamento della rete invece prevede che i valori del gradiente vengano passati all'algoritmo di discesa del gradiente che li sfrutterà per aggiornare i pesi di ciascun neurone. L'aggiornamento dei pesi viene fatto usando una costante detta **learning rate**.
L'addestramento è svolto in cicli con due possibili modalità cioè quella stocastica e quella

di gruppo: nella prima, ogni passo di propagazione in avanti è seguito immediatamente da un passo di aggiornamento, mentre nella seconda viene effettuata la propagazione per tutti gli esempi del training set e dopo viene fatto l'aggiornamento. Questo secondo approccio porta direttamente al risultato finale ma risulta essere infattibile o comunque computazionalmente esagerato.
Si è trovato in seguito un buon compromesso grazie all'uso di **mini-batch** cioè dei piccoli insiemi casuali di dati sui cui andare ad allenare la rete.

2.7.1 Loss Function ed Overfitting

Gli algoritmi di apprendimento supervisionato necessitano di uno strumento per misurare la qualità delle predizioni in funzione dei parametri del modello: le **Loss Fuction** o **funzioni di perdita** che sono state presentate in precedenza come una delle scelte libere affidate all'utente quando progetta una architettura di rete neurale e che costituiscono il metodo per valutare le prestazioni della rete in fase di training. Esistono diverse loss function: comunemente ciascuna di esse misura la discrepanza tra valori reali e valori predetti, quindi in base alla scelta fatta si potranno avere diversi risultati.
La loss function L deve essere scelta in base al tipo di task da affrontare poiché non è detto che quella scelta sia adatta per il numero di classi scelte; permette inoltre di creare una sorta di parametro di regolarizzazione che penalizza alcuni errori e conseguentemente modifica ciò che il modello apprende, dal momento che la rete in fase di apprendimento lavora sempre e solo sul training set. Viene fuori in questa fase il rischio di apprendere valori che vanno a interpolare le features estratte rischiando di cadere nella tendenza di memorizzare il training set. Questa condizione appena evidenziata è detta **overfitting**: si va ad inseguire l'andamento dei dati di training.

2.7.2 Discesa del Gradiente

Il Gradient Descent [18] o Discesa del Gradiente è un algoritmo mirato ad ottenere la stima dei pesi: viene minimizzata una funzione obiettivo $J(\Theta)$ formata da N parametri (Θ) e viene aggiornato il valore dei parametri in base alla differenza con il gradiente negativo di $J(\Theta)$ rispetto al parametro considerato. L'aggiornamento del parametro viene fatto attraverso un valore η chiamato **learning rate**. Sarà quindi in parole povere sfruttata una funzione $J(\Theta)$ allo scopo di arrivare ad un valore di massimo o di minimo in cui fermarsi. I modi con cui questo algoritmo si può eseguire sono: la **Batch Gradient Descent (BGD)** [18], o Discesa del Gradiente a Batch, che comporta la discesa del gradiente su tutto il Training set; lo **Stochastic Gradient Descent (SGD**) [19], o Discesa del Gradiente Stocastica, che esegue la discesa del gradiente per ogni elemento del Training set e aggiorna il suo valore volta per volta. Infine, si può citare la **Mini Batch Gradient Descent (MBGD)** [18], o Discesa del Gradiente a Mini Batch, che è una via di mezzo fra la SGD e la BGD, in quanto effettua degli aggiornamenti ai parametri della funzione con set ridotti, evitando l'uso di tutto il dataset o di singoli valori raggiungendo velocemente i valori di minimo e massimo.

2.7.3 Batch-Normalization

La tecnica di Batch Normalization [20] è una tecnica che permette di migliorare velocità, performance e stabilità di una rete neurale. È utilizzata per normalizzare l'input e sembra proprio che possa risolvere il problema dello shift covariato interno, nel quale i parametri

di inizializzazione e i cambi di distribuzione degli input portano al cambio del learning rate.
Nello specifico, per spiegare bene lo shift di covarianza è possibile presentare questo esempio: passando l'immagine di un gatto colorato ad una rete è chiaro che ci siano problemi di riconoscimento rispetto a quando viene dato in input un tradizionale gatto, visto che la rete ha imparato da un determinato tipo di immagine.
Questa tecnica riduce il problema indicato permettendo ad ogni layer di imparare autonomamente ed in maniera indipendente rispetto agli altri layer.

2.7.4 Dropout

Il Dropout [21] si riferisce all'ignorare alcune unità neurali durante le fasi di allenamento. Appare come una forma di regolarizzazione che agisce "sganciando" alcuni nodi dalla rete. Tutto ciò viene fatto per ridurre l'overfitting o comunque per prevenirlo. L'applicazione del dropout produce ad ogni iterazione una diversa rete ridotta del modello di partenza, composta da quei nodi che sono sopravvissuti al processo di drop.
Di conseguenza viene a crearsi una sorta di combinazione di modelli che tende quasi sempre a migliorare la performance, riducendo l'errore di generalizzazione. L'idea quindi, è quella di utilizzare una singola rete neurale completa, i cui pesi sono le versioni ridimensionate dei pesi calcolati in precedenza.
Il Dropout è implementato per ogni layer all'interno della rete neurale e può essere usato quindi con tutti i tipi di layer ma non nel layer di output.

Capitolo 3

Lo Stato Dell'Arte delle Reti Neurali

3.1 Introduzione allo stato dell'arte

Questo capitolo sarà totalmente diverso rispetto precedente che offriva una panoramica riassuntiva e delle nozioni tecnico teoriche sul mondo del Machine Learning e delle reti neurali; adesso saranno passate in rassegna quelle che sono state le famiglie di reti e i relativi modelli progettati negli ultimi decenni, fino ad arrivare ai modelli ancora oggi utilizzati. Vale anche qui la doverosa precisazione sul termine "modello", che come già detto, è inteso come la distribuzione di una specifica architettura che appartiene ad una determinata famiglia: ad esempio, parlando della famiglia ResNet, sarà usato il termine modello, quando si parlerà di una versione della rete appartenente alla suddetta famiglia. Verrà inoltre fatta una suddivisione in merito ai task per cui sono state progettate le reti che si vanno ad analizzare: nello specifico classificazione, localizzazione, segmentazione di oggetti in immagini e tracking di oggetti nei video. Un'ennesima suddivisione è stata fatta in base al fatto che siano state progettate per lavorare in ambito mobile o meno.
Si partirà da una rassegna storica in modo da avere una base da cui partire per analizzare ciò che viene offerto al momento come "stato dell'arte" mentre nella parte conclusiva del capitolo invece, saranno toccati i video e l'object tracking nei video e un tipo di approccio ibrido che si pone a metà strada tra una architettura vera e propria ed una API che è YOLO.

3.2 The ImageNet Large Scale Visual Recognition Challenge (ILSVRC)

Figura 3.1: Logo del ILSVRC.

Nell'ambito della ricerca di questa tesi è doveroso partire da un evento scientifico che ha dato via alla grande macchina dell'evoluzione dell'**Image Recognition**: **"The ImageNet Large Scale Visual Recognition Challenge (ILSVRC)"** [22]. Fu una sfida la cui prima edizione si tenne nel 2010, e che si tiene ancora oggi con cadenza annuale, il cui obiettivo principale era ed è valutare algoritmi per il riconoscimento e la classificazione degli oggetti nelle immagini su larga scala. Lo scopo iniziale della challenge era stimare il contenuto delle immagini, fornendo una annotazione automatica. In particolare, questa operazione venne effettuata su un sottoinsieme di immagini: venne utilizzato il dataset ImageNet che disponeva già delle necessarie annotazioni relativamente alle categorie di oggetti riconoscibili.
La base della sfida era l'uso obbligatorio di questo set pubblico di immagini annotato per comparare degli algoritmi di riconoscimento mentre la competizione e il workshop, conseguente alla competizione, permettevano di discutere riguardo alle migliorie ottenute nell'anno in termini di risultati e prestazioni.
Le annotazioni del dataset in particolare ricadono in due categorie: **image-level annotation** in riferimento alla presenza o no di un oggetto nell'immagine quindi ad esempio (c'è una macchina nell'immagine o non c'è una tigre nell'immagine) e **object-level annotation** con delle annotazioni specifiche sull'oggetto relativamente a posizione e dimensioni in pixel di un determinato oggetto nell'immagine (c'è una macchina alla posizione centrale 20, 25, di altezza 30pixel e larghezza 20 pixel).
La sfida, andando di pari passo con l'evoluzione degli algoritmi e delle tecnologie, ha mutato/aumentato negli anni il numero di task da performare:

- 2010-2014: Produzione come detto di una lista di categorie di oggetti presenti.
- 2011-2014: Produzione di una lista di categorie di oggetti, con in aggiunta l'annotazione della posizione all'interno di un box e delle dimensioni del SINGOLO oggetto categorizzato (**SINGLE OBJECT LOCALIZATION**).
- 2013-2014: Produzione di una lista di categorie di oggetti, con in aggiunta l'annotazione della posizione all'interno di un box e delle dimensioni di tutti gli oggetti della relativa categoria (**OBJECT DETECTION**).

Ogni edizione è stata caratterizzata da nuove proposte algoritmiche e da nuove migliorie nello stato dell'arte: saranno passate in rassegna le edizioni che hanno segnato una svolta migliorativa delle implementazioni. L'anno cruciale fu il 2012.
Nella prima edizione del 2010 [23] (solo in questa edizione) venne fornito un set di immagini, così che i partecipanti potessero allenare i propri algoritmi e conseguentemente annotare le immagini test.
Queste immagini annotate venivano poi sottomesse su un server valutativo dai partecipanti stessi e al termine della competizione i risultati della sfida venivano rivelati al pubblico: coloro che avevano conseguito i risultati migliori erano invitati a condividerli nella **International Conference on Computer Vision (ICCV)** o nella **European Conference on Computer Vision (ECCV)** che si tenevano ad anni alternati.
Nel 2010 i vincitori del NEC team (Lin et al.,2011) usarono l'algoritmo **SIFT** [24] e l'algoritmo **LBP** [25] sfruttando due codifiche di Classificazione non lineare e una **SVM (Support Vector Machine [27]) stocastica**.
Di seguito saranno esposte le tecniche citate, fornendo le relative referenze per permettere un maggiore approfondimento:

- **SIFT (Scale Invariant Feature Transform)** [24]: è un algoritmo che estrae alcuni "feature point", cioè i punti interessanti di alcune immagini di riferimento, al fine di fornire una descrizione delle caratteristiche dell'oggetto. L'algoritmo si compone essenzialmente di una prima parte in cui vengono riconosciuti i keypoint e poi localizzati, seguita da una parte in cui verrà creato una sorta di orientamento dei punti rispetto ai bordi dell'immagine, che è il preludio allo step finale nel quale verrà creato il descrittore delle features.
  Uno degli elementi fondamentali di questo algoritmo è il **DoG (difference of Gaussians)** che implica l'uso di una versione sfocata dell'immagine originale, che verrà sottratta da una meno sfocata dell'originale, al fine di reperire proprio nelle immagini i cosiddetti feature point.

- **LBP (Local Binary Pattern)** [25] [28]: è un semplice ma efficiente descrittore per le texture dell'oggetto. È stato molto utilizzato in applicazioni di riconoscimento di volti umani in 2D e venne introdotto da Ojala [26].
  Il funzionamento dell'operatore è molto semplice: a ciascun pixel dell'immagine viene assegnato un valore binario, dipendente dalla valorizzazione dei pixel che stanno nell'intorno preso in considerazione (ad esempio 3x3). Se si prendesse in considerazione un pixel campione centrale "pc" da valutare e un pixel p nell'intorno considerato, se p assumesse un valore superiore o uguale a quello di pc allora a p verrebbe assegnato il valore 1 altrimenti il valore 0.
  Mettendo in sequenza i valori binari ottenuti si ottiene un numero binario finale, il cui numero di cifre dipende dalla dimensione dell'intorno. L'inconveniente di questa prima versione dell'algoritmo era proprio la dimensione dell'intorno: all'inizio era prefissato a 3, mentre poi successivamente venne reso libero e l'operatore LBP venne esteso per gestire intorni di dimensioni variabili.
  Si passò poi anche a considerare la gestione non più di un intorno quadrato ma bensì anche di un intorno circolare inserendo ancora due elementi: il numero di punti campione P e il valore del raggio R. Il procedimento risultò essere lo stesso ma permise di valutare un numero di campioni più ampio.

- Le **SVM cioè Macchine a Vettori di Supporto [27]** sono dei modelli di apprendimento supervisionato utilizzabili sia per scopi di apprendimento che per scopi di classificazione.
  Ottengono i maggiori risultati nei problemi di classificazione binaria ma nonostante ciò sono lo strumento più utilizzato per la classificazione di pattern. L'idea alla base è la seguente: date due classi di appartenenza e dato un insieme di esempi detto training set, ogni esempio viene etichettato con una delle due classi indicate e successivamente mediante un algoritmo di addestramento per le SVM viene creato un modello che assegna automaticamente un possibile nuovo esempio ad una delle due classi.

Nel 2011 il team XRCE, già menzionato nella precedente edizione (2010) per avere proposto il **Vettore di Fisher [29]** in versione migliorata ed adattata alla classificazione su larga scala, vinse stavolta la gara di Classificazione proponendo un **vettore di Fisher** ottimizzato per la classificazione su larga scala [30], riproponendo ciò che avevano proposto nella precedente edizione ma affiancandolo ad una **Compressione PQ (Product Quantization** [31] [32] [33] e ad una **Semplice Macchina di Apprendimento Artificiale**: la **One-vs-all linear Simple Vector Machine (one vs-all linear SVMs)** [34]. Il vettore di Fisher risultò infatti essere molto performante a livello computazionale, poichè forniva eccellenti risultati con i classificatori lineari e soprattutto permetteva di essere compresso con una perdita minima.
La combinazione di questi 3 elementi permise al team di ottenere dei risultati ottimali per il task di classificazione in quell'annata.
Per quanto riguarda il task di single-object localization i vincitori furono i componenti del team UvA, che sfruttando un approccio di **ricerca selettiva** [35] campionarono densamente [36] e quantizzarono i descrittori SIFT dei diversi colori; poi raggruppando i risultati con una piramide spaziale [38] e classificando con una **"histogram intersection kernel SVM"** [39] allenata su una GPU [40] riuscirono ad ottenere delle sostanziali migliorie prestazionali.
La **ricerca selettiva** è uno dei metodi maggiormente utilizzati per generare possibili regioni di posizionamento degli oggetti: viene ottenuto una sorta di vettore per ogni papabile regione, sul quale viene applicata una piramide spaziale multivello al fine di avere una compattazione dei prodotti, dal momento che avevano dimensioni eccessive ma anche per mantenere sempre la qualità visiva dell'immagine.
Come detto però l'edizione del 2012 è quella che diede la svolta dal momento che entrarono in scena tecniche ancora oggi molto utilizzate.
In particolare, il SuperVision team, allenò una rete neurale convolutiva su valori RGB con 60 milioni di parametri, sfruttando anche l'uso di una GPU ed utilizzando la funzione di attivazione ReLU [41] e la tecnica del Drop-out per ridurre l'overfitting [42].
Questa fu la prima rete neurale presentata nel campo di nostro interesse e tra i primi risultato tangibili è sottolineabile l'abbassamento della percentuale di errore dal 26% al 15%, dato che denota un grande miglioramento nelle performance di accuratezza di predizione.
La loro architettura conteneva 5 strati di convoluzione, intervallati da degli strati di pooling e da 3 strati completamente connessi. Ciascuno strato è inoltre suddiviso in due sottostrati per permettere l'elaborazione su due GPU.
Con certezza si può dire che questa fu l'edizione che via via spinse sempre di più verso l'uso delle reti neurali per la classificazione e la localizzazione e soprattutto a partire da quel momento iniziarono ad essere presentati e studiati molteplici modelli architetturali differenti tra loro che iniziarono a dare predizioni sempre più accurate, performance migliorate e soprattutto costi computazionali sempre ridotti, cosa che non aveva permesso in precedenza l'implementazione pratica delle reti neurali.

Così nell'edizione del 2013 la maggior parte dei partecipanti utilizzò reti di convoluzione e tra tutti i team iscritti al contest, i vincitori furono i componenti di Clarifai, con una rete progettata mediando diverse reti convoluzionali multi-layer. Le architetture furono sfruttate usando la tecnica di visualizzazione di Zeiler e Fergus [43] e addestrate su GPU con la tecnica del dropout [44].
Nel task di localizzazione di un singolo oggetto, il vincitore fu il progetto chiamato "OverFeat", basato su un framework mirato all'utilizzo di reti convoluzionali per la classificazione, localizzazione e rilevamento mediante finestra multi-scala scorrevole [45].
In questo anno venne anche introdotto il task di rilevamento di oggetti multipli, il cui vincitore fu, ancora una volta, il team UvA dell'Università di Amsterdam che, utilizzando un nuovo metodo efficiente di codifica, progettato nuovamente da Van de Sande come nel 2011, riuscì a campionare densamente i descrittori dei colori per poi raggrupparli in una struttura di ricerca selettiva basata su una piramide spaziale multilivello [46] [47].
Il 2014 fu una annata di grandi risultati dato che si dimezzò l'errore nella classificazione rispetto alla precedente edizione e di conseguenza raddoppiò anche la media nel riconoscimento corretto di oggetti; venne permesso l'uso di dati esterni per allenare le reti e per ciascuno dei 3 task quindi vennero proposte due competizioni: una tradizionale e una quindi con dati di allenamento esterno.
Balzò agli onori la **"GoogLeNet"** che vinse il contest di classificazione grazie all'introduzione dei layer "Inception" [48], mentre nel contest di localizzazione è invece la rete **VGG** [49] ad ottenere i migliori risultati. I dettagli su queste due architetture saranno analizzati in seguito nella sezione descrittiva sui modelli conosciuti.
Nel 2015 le prestazioni sul dataset Imagenet vennero ancora migliorate e vennero introdotti due ulteriori task:

- il rilevamento di oggetti in filmati, con l'uso di sole 30 categorie, sottoinsieme delle 200 categorie della competizione di rilevamento di oggetti.
- la classificazione di scene, organizzata dal MIT Places Team basata su 401 categorie.

A vincere in tutte le categorie di riconoscimento, rivelazione e classificazione di oggetti è il team MSRA (Microsoft Research Asia), presentando una rete di convoluzione molto semplice, a cui venne applicato per la prima volta il concetto dell'apprendimento residuale: la **"ResNet"** [50].
Il 2016 è da considerare un anno rivoluzionario: nei contest di Object Localization, Object Classification ottenne il primo posto un team cinese il cui nome è Trimps-Soushen dove Trimps sta a significare "Third Research Institute of Ministry of Public Security", organo di sviluppo tecnologico lanciato nel 1978 a Shangai mentre Soushen è il vero e proprio team che significa letteralmente "God of search"; ma le vere innovazioni furono due e non furono implementative: non venne pubblicato alcun paper e nessun report tecnico sul loro operato dal momento che si trattava dello stato dell'arte su più task e dal momento che non ci furono grandi innovazioni tecniche pur riuscendo a scendere ad un rate di errore sotto il 3%, quando l'occhio umano riesce a raggiungere solo il 5%.
Per la localizzazione vennero usati i modelli pre-allenati di Inception v3 [51], Inception v4 [52], Inception-ResNet-v2 [53], Pre-Activation ResNet-200 e Wide ResNet, cercando di aggiustare alcuni parametri di training e testing: capirono che ciascuno di questi modelli di rete otteneva risultati eccellenti in alcune categorie di oggetti ma era carente su altre categorie di oggetti.
La vera e propria intuizione che portò alla vittoria e ai grandi risultati raggiunti fu l'uso della Region Fusion con le Faster R-CNN [53] [54].

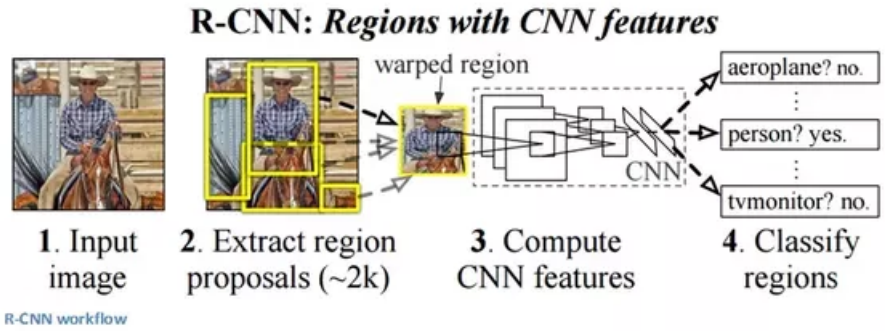


Figura 3.2: Approccio delle Architetture Region Based. Fonte: [55].

Nell'edizione del 2017 invece la fondazione SENet partecipò con eccellenti risultati alla competizione abbassando i risultati percentuali di predizione con un nuovo modello architetturale chiamato **"Squeeze and Excitation Networks** [56] [57]: il loro obiettivo fu quello di migliorare la potenza di una rete modellando esplicitamente le dipendenze tra i canali convolutivi e per fare ciò proposero un meccanismo di "feature recalibration" attraverso cui imparare ad usare le informazioni globali per selezionare le feature interessanti eliminando quelle poco utili.
La rete proposta era la SeNets che in sintesi aggiungeva parametri ad ogni canale del blocco convolutivo, così che si potessero aggiustare i pesi di ogni mappa. Quello che bisogna considerare è che i pesi della rete per ogni canale sono creati in modo eguale quando si ottengono le relative feature map. Il blocco SE che implementarono opera sull'input del blocco convolutivo e sul numero di canali, facendo un average pooling; il tutto è seguito da un fully connected layer con una successiva ReLU che aggiunge la non-linearità. Infine, viene aggiunto un secondo fully connected layer seguito da una attivazione Sigmoidale e viene pesata ogni feature map del blocco convolutivo.

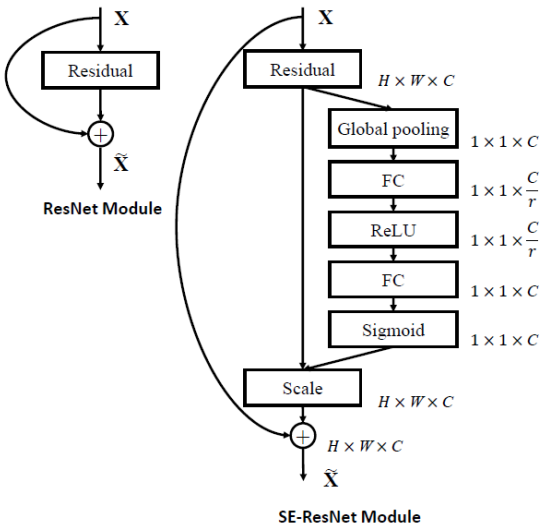


Figura 3.3: A destra il blocco Residuale tradizionale mentre a sinistra il blocco residuale inserito nella Squeeze and Excitation Net. Fonte ed approfondimento: [57].

3.3 Architetture dei Modelli CNN

Le Cnn vennero proposte per la prima volta da **Fukushima** nel 1988 [58], ma non vennero praticamente usate per via del grosso carico computazionale che richiedeva il processo di training. Così negli anni 90 Yann LeCun, applicando un algoritmo di apprendimento basato sul gradiente, portò avanti un nuovo progetto di studio sviluppando la sua **LeNet [59]** ed ottenendo dei successi nell'ambito del problema della classificazione delle cifre scritte manualmente.
Man mano, col tempo, ci si accorse che con varie migliorie, si potevano ottenere sempre migliori prestazioni usando le CNN, arrivando a definirle come le migliori strutture per processare immagini in 2D e 3D, proprio perché il loro comportamento assimilava sempre meglio il comportamento dell'apprendimento visivo umano.
Sostanzialmente però fino al 2010 non si era riusciti ancora ad avere delle implementazioni vere e proprie a causa della limitata capacità di calcolo computazionale e di memoria di cui si poteva disporre.

3.3.1 1994: LeNet5

La sua architettura nel 1994 venne conosciuta come **LeNet5** e venne considerata esattamente come primo vero e proprio modello convolutivo; fu il risultato di diversi tentativi, compiuti da LeCun, a partire dal 1988.
Da sottolineare come ai tempi non essendoci GPU dedicate, le CPU erano parecchio lente quindi fare training era complicato e il salvataggio dei dati era tanto importante quanto complicato.
La struttura si componeva di due layer convoluzionali, due sottostrati di campionamento posti dopo il livello convolutivo e 2 layer fully connected seguiti da un layer di output con connessione Gaussiana.

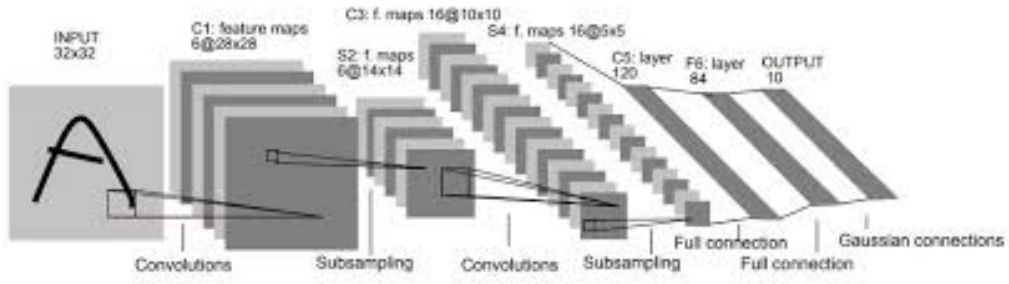


Figura 3.4: Classica architettura LeNet5. Fonte: [60].

Nel momento in cui l'hardware cominciò ad essere sempre più efficiente, le CNN iniziarono ad avere credibilità anche a livello implementativo divenendo il caposaldo per i ricercatori e come già visto nella sezione introduttiva, grazie ad **Alex Krizhevesky [62]** che insieme al suo gruppo di studi vinse l'ILSVRC 2012, venne riscritto lo stato dell'arte del deep learning.

3.3.2 2012: AlexNet

La struttura prevedeva un primo livello che si occupasse della convoluzione e del **Max Pooling**, implementato con LRN (Local Response Normalization): erano stati usati 96 differenti filtri recettivi, di dimensione 11x11; nel secondo livello con filtri 5x5 era svolta la medesima operazione. Nel terzo, quarto e quinto erano usati filtri 3x3 ma con rispettivamente 384, 384, 296 feature map per l'operazione di convoluzione che era seguita dalla funzione di Relu. Infine, i due fully layer erano usati con funzione softmax.

AlexNet Vs LeNet

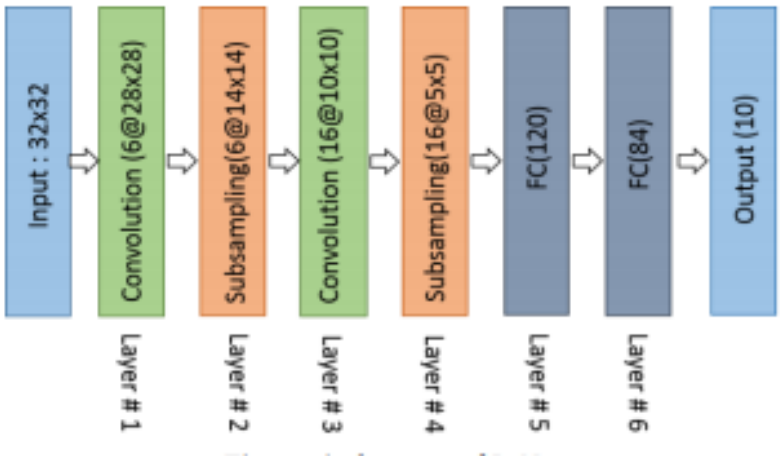


Figura 3.5: Blocchi LeNet5.Fonte ed approfondimenti: [61].

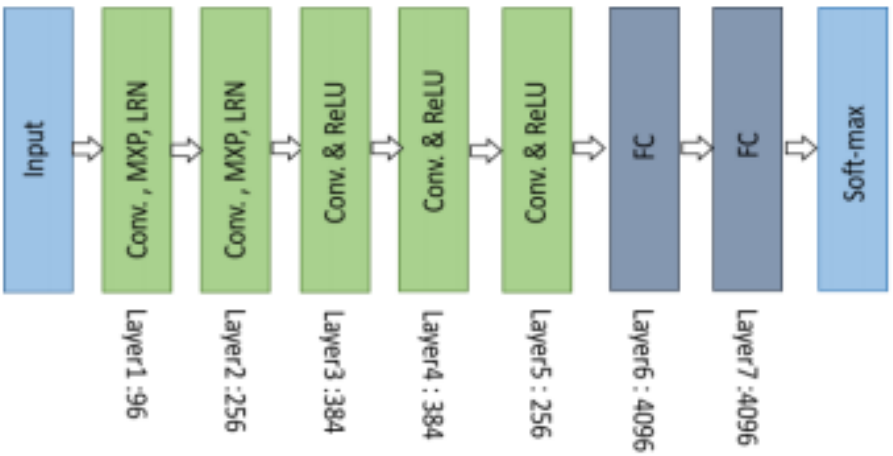


Figura 3.6: Blocchi AlexNet. Fonte: [61].

3.3.3 2013: ZFNet

Nel 2013 Matthew Zeiler and Rob Fergue vinsero l'ILSVRC con una CNN che estendeva la **AlexNet**, chiamata **ZFNet [43]** il cui nome derivò proprio dal nome dei due scopritori. Il vincitore del contest di classificazione fu il team Clarifai, il cui fondatore e CEO era Zieler. Questo team fece solo delle modifiche minime sulla ZFNet che è stata considerata come rete vincitrice del contest. Anche questa rete è costosa a livello computazionale come la AlexNet, ma è stata creata ottimizzando la precedente con un kernel sul primo livello 7x7, piuttosto che 11x11 per ridurre il numero di filtri e il numero di parametri e soprattutto per ridurre la profondità del primo layer convolutivo da 4 a 2.
La ZFNet ha di contro migliorato in maniera significativa l'errore di predizione rispetto alla AlexNet.

3.3.4 2013: Network in Network

Una rete, apparentemente simile alle precedenti, ma che presenta diversi cambiamenti è la NiN cioè la **Network in Network [63]**: introduce l'uso della **MLP (Multilayer Perceptron)** ed è quindi uno dei primi prototipi di convoluzione multilivello, nel senso che dopo il livello tradizionale di convoluzione venne usato un livello aggiuntivo per combinare le feature tra un livello e il successivo.
Venne implementata con filtri 1x1 in modo da aumentare la non linearità del modello ed incrementare la profondità della rete. Questa idea venne ripresa in reti che divennero molto più famose come le ResNet, le Inception e le relative derivate. La Nin Introduce anche la **Global Average Pooling (Gap)** al posto dei layer fully connected, riducendo ancora il numero di parametri: la struttura con i Gap veniva cambiata generando alla fine un vettore di feature molto più piccolo quindi generando mappe di feature di dimensione ridotta.

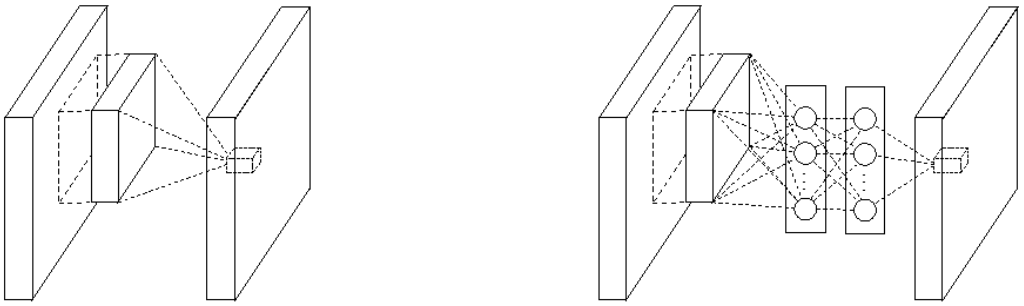


Figura 3.7: la figura a destra rappresenta un layer tradizionale mentre quella a sinistra un multilayer perceptron. Fonte: [63].

3.3.5 2014: VGG-Net

Le **VGG-NET** [**49**], il cui nome deriva da Visual Geometry Group, si affacciarono nel 2014 come progetto di spicco per l'ILSVRC per il task di classificazione pur vincendo il contest per quanto concerne il task di localizzazione.
Posero agli occhi di tutti quella che era a loro avviso la svolta concettuale per ottenere migliorie: la profondità della rete. Fu il primo anno in cui un modello di rete convolutiva ottenne un rate di errore sotto il 10%. Furono tra le prime reti che sfruttarono l'idea che usare filtri 3x3, quindi filtri più piccoli, ma ripetuti in sequenza, permettendo di ottenere gli stessi risultati ottenuti usando filtri recettivi molto più grandi: idea in forte contrapposizione con le AlexNet. Chiaramente questo aumentò il numero di filtri.
Ottennero un grosso risparmio computazionale visto che diminuì il numero di parametri, pur producendo una mappatura migliore tra le immagini e le etichette di categoria di classificazione.
Per la classificazione venne progettata una rete che di base proponeva due livelli convolutivi basati sulla funzione di attivazione RELU con un layer SoftMax finale ottimizzato per l'operazione di classificazione.
Per la localizzazione, i bounding box sono rappresentati con vettori in 4-D riferiti alle coordinate di centro, altezza e larghezza. La regressione logistica fu rimpiazzata con la funzione di perdita euclidea.
I Modelli proposti VGG 11, 16 e 19 differiscono tra loro per il numero di layer, con rispettivamente 8, 13, 16 layer convolutivi.
La VGG 19 divenne il modello più costoso a livello computazionale in quel momento.

VGG-Net vs AlexNet

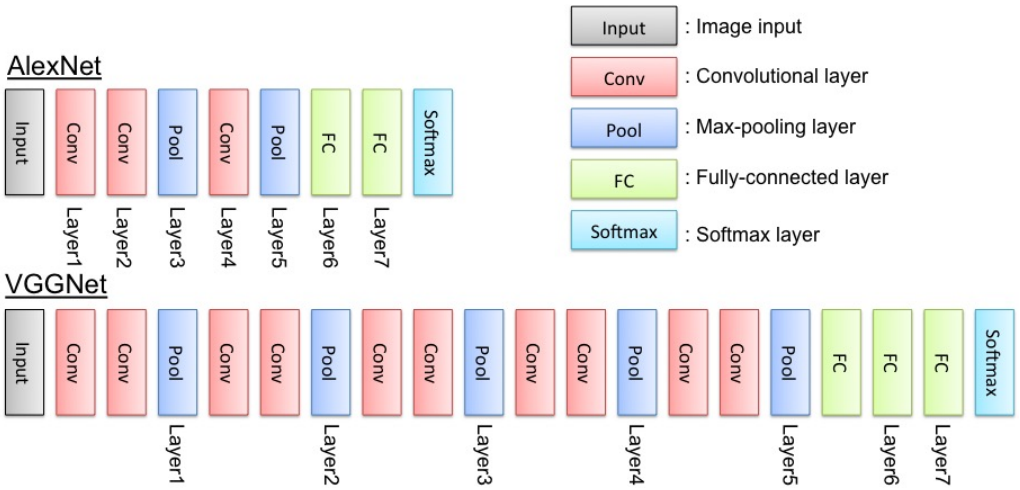


Figura 3.8: Confronto ad alto livello tra le due architetture e di quanto esposto nella precedente sezione. Fonte: [64].

3.3.6 2014: GoogLeNet (Inception V1)

Il 2014 fu un altro anno epocale poiché il vincitore fu **Chrstian Szegedy** di Google [51] [52] [48] che si era prefissato l'obiettivo di ridurre la complessità computazionale rispetto alle già esageratamente costose vecchie CNN: introdusse e propose l'**Inception Layer**, che disponeva di campi ricettivi variabili, realizzati mediante l'uso di kernel di dimensioni diverse e mirati a catturare le correlazioni tra i pattern con delle specifiche operazioni.
Sostanzialmente quindi questa fu la prima architettura Inception, il cui nome è un chiaro riferimento all'omonimo film. L'idea fu quella di combinare filtri 1x1, 3x3, 5x5 tra di loro all'interno del blocco Inception.

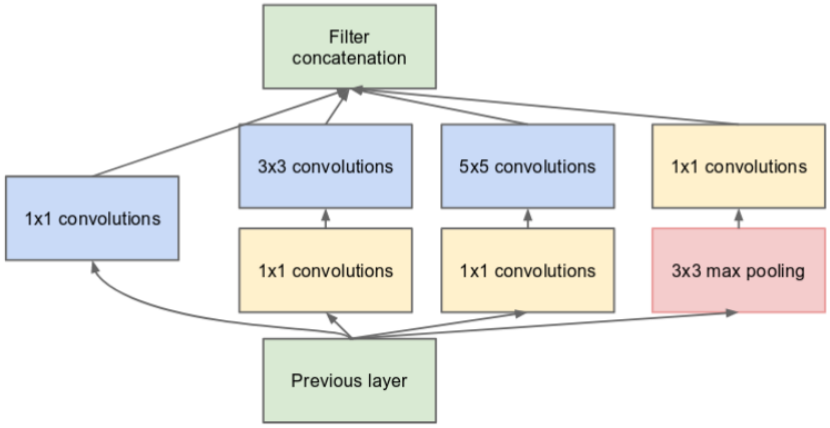


Figura 3.9: Blocco Inception nella sua versione, con le riduzioni della dimensione. Fonte ed approfondimenti: [65].

Nel blocco della figura precedente si notano 4 percorsi paralleli: il primo usa un layer 1x1 come ripetitore dell'input, il secondo un 1x1 seguito da un layer 3x3, il terzo un layer 1x1 seguito da uno 5x5 e l'ultimo usa un layer 3x3 unito ad un 1x1. I percorsi centrali usano il blocco 1x1 sull'input, per ridurre il numero di canali di input e ridurre la complessità. Alla fine l'output di ogni path verrà concatenato e dato in input al layer successivo. Nell'Inception Block ciò che risulta essere customizzabile sono il numero di canali di output per layer, in modo da gestire autonomamente la complessità.
L'applicazione di questo blocco permise grossi miglioramenti prestazionali e di numero di parametri e portò alla riduzione dell'overfitting.
Per la prima **GoogleLeNet** venne utilizzato uno stack totale di 9 blocchi inception mentre le funzioni di Max Pooling poste tra i blocchi Inception permisero di ridurre la dimensionalità: la struttura nella prima parte era identica alla AlexNet e alla LeNET, lo stack dei blocchi centrali è derivato dalle VGG ed infine prima dell'uscita il Global Average Pooling sostituì l'uso di uno stack di layer fully connected. Il totale finale risultò essere di 22 layer più 5 layer di pooling. Questa struttura viene solitamente chiamata **Inception v1**.

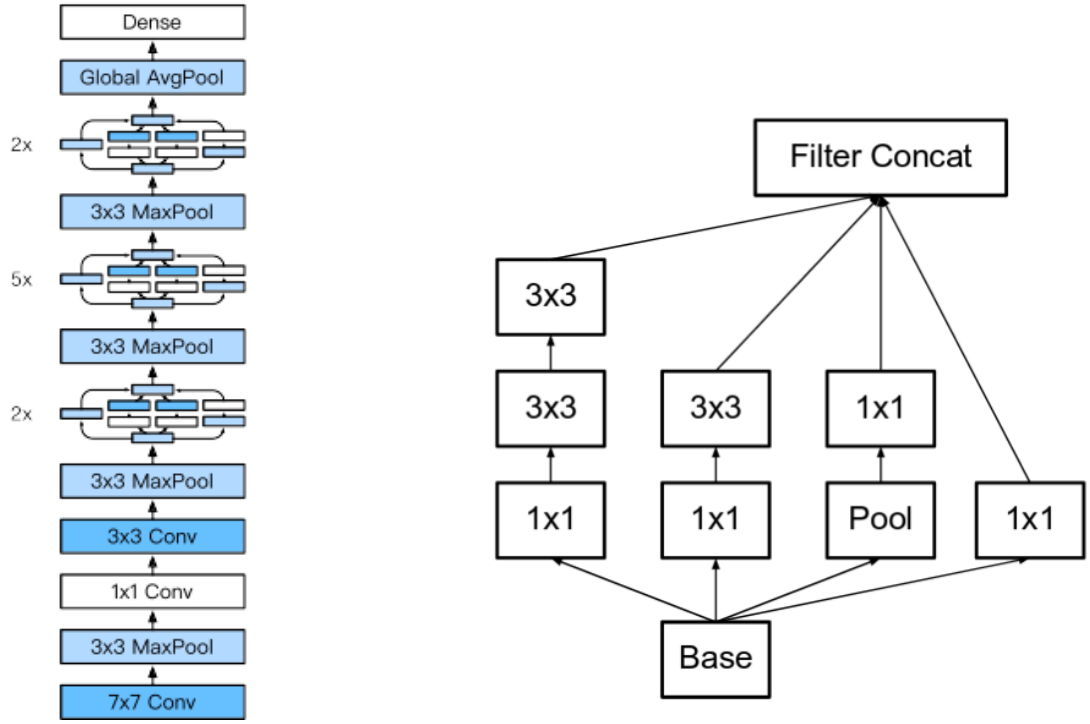


Figura 3.10: A destra Architettura Inception V1, a sinistra blocco Inception V2. Fonte ed approfondimenti: [65].

3.3.7 2015: Inception V2 & Inception V3

Diventò necessario ridurre l'impatto della computazione nei layer dei blocchi Inception: l'intuizione fu che le reti neurali lavorano meglio quando la convoluzione non altera le dimensioni dell'input in modo drastico. Riducendo le dimensioni però si creava il collo di bottiglia (**BottleNeck Layer**) che scaturiva dal fatto che le convoluzioni erano parallele e gran parte delle operazioni era concentrata in quel layer.
Chrstian Szegedy e il suo team di ricercatori così iniziarono a proporre delle migliorie alla precedente struttura arrivando a sviluppare la versione 2 e la versione 3 della rete Inception.
Il team ebbe una importante intuizione: sostituire layer più grandi, come quelli 5x5 o 7x7, con layer consecutivi di dimensioni inferiori.
Così ad esempio, nella Inception v2, cavalcando l'idea appena esposta venne sostituito il layer 5x5 con due layer 3x3 mentre nella versione 3 sostituirono il layer 7x7 con tre layer 3x3.
Evidenziarono così altre possibili migliorie notando che i classificatori ausiliari non contribuivano molto nella parte finale del processo di training ed introdussero la **Batch Normalization** o il **Dropout** per regolarizzare queste funzioni.
Entrambe vennero introdotte nella versione 3 insieme al **RMSPropOptimizer** e al **Label Smoothing**, un componente per regolarizzare l'overfitting quando la rete inizia ad essere troppo precisa/fidata nella predizione rispetto una specifica classe.

3.3.8 Inception-V4

Inception-v4 è una evoluzione delle precedenti, molto più semplificata e con più moduli della V3. Purtroppo, fin da subito non vennero date grosse spiegazioni riguardo alle nuove introduzioni architetturali.
In particolare, si è evidenziato che l'immagine passa prima in un layer detto Stem, per poi finire in 3 tipi diversi di livelli Inception chiamati A, B, C.
L'Inception v4 introdusse i "Reduction Blocks" utili a cambiare l'altezza e la larghezza dei moduli.

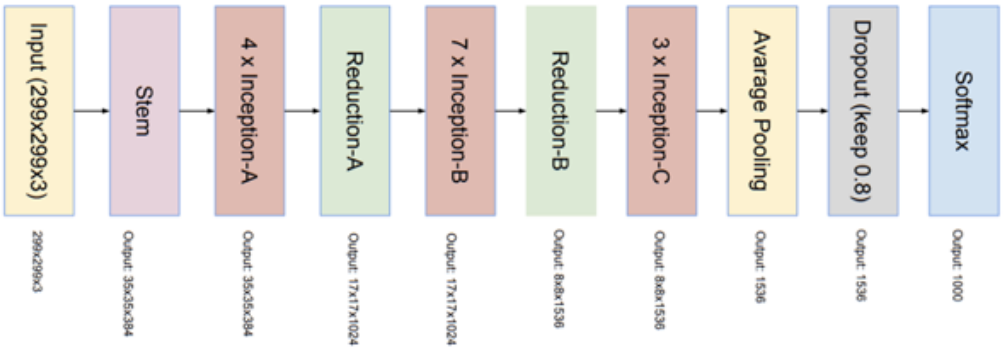


Figura 3.11: Architettura Inception V4. Fonte ed approfondimenti: [65] [66] [68].

3.3.9 2015: ResNet

Residual Network [50]: e' il 2015 e Kaming progettò una rete rivoluzionaria che cambiò qualcosa che era stata alla base delle precedenti proposte.
ResNet nacque dall'osservazione che con l'aumento di livelli ci potesse essere il rischio di imbattersi in peggioramenti della rete. Intuitivamente, reti neurali più profonde non dovrebbero performare peggio di quelle poco profonde, o almeno non durante l'allenamento quando non vi è alcun rischio di overfitting. Tuttavia, al crescere della profondità della rete questo non è sempre vero.
Gli sviluppatori di ResNet ricondussero questo problema all'ipotesi che le mappature dirette sono difficili da allenare e proposero un rimedio, ovvero l'uso del blocco residuale: l'idea fu che la rete ora potesse imparare principalmente le differenze tra i layers in ingresso e in uscita dal blocco. I layer delle reti proposte variarono allora tra "34", "50", "101", "152",
"1202". Ad esempio, la più popolare cioè la ResNet50 disponeva 49 layer convolutivi e di uno fully connected.
Le ResNet quindi sono delle reti **"feed forward"** con una **"residual connection"**: quest'ultima, costruita con diversi tipi di blocchi di resto in base al tipo diverso di architettura scelta. Prima di allora era molto diffuso il problema dell'annullamento del gradiente, la cui discesa, data dalla minimizzazione della funzione di errore, si riduce esponenzialmente attraverso la retro propagazione degli strati precedenti. In sostanza, il percorso lungo gli strati precedenti rendeva gli errori talmente piccoli da non permettere alla rete di apprendere. Con le ResNet iniziarono a vedersi reti con innumerevoli strati caratterizzati da un elevato grado di accuratezza.
Questo diede il via ai nuovi progetti: la versione più ampia di residual network venne proposta da Zagoruvko nel 2016 [67], anno nel quale venne proposta anche una versione

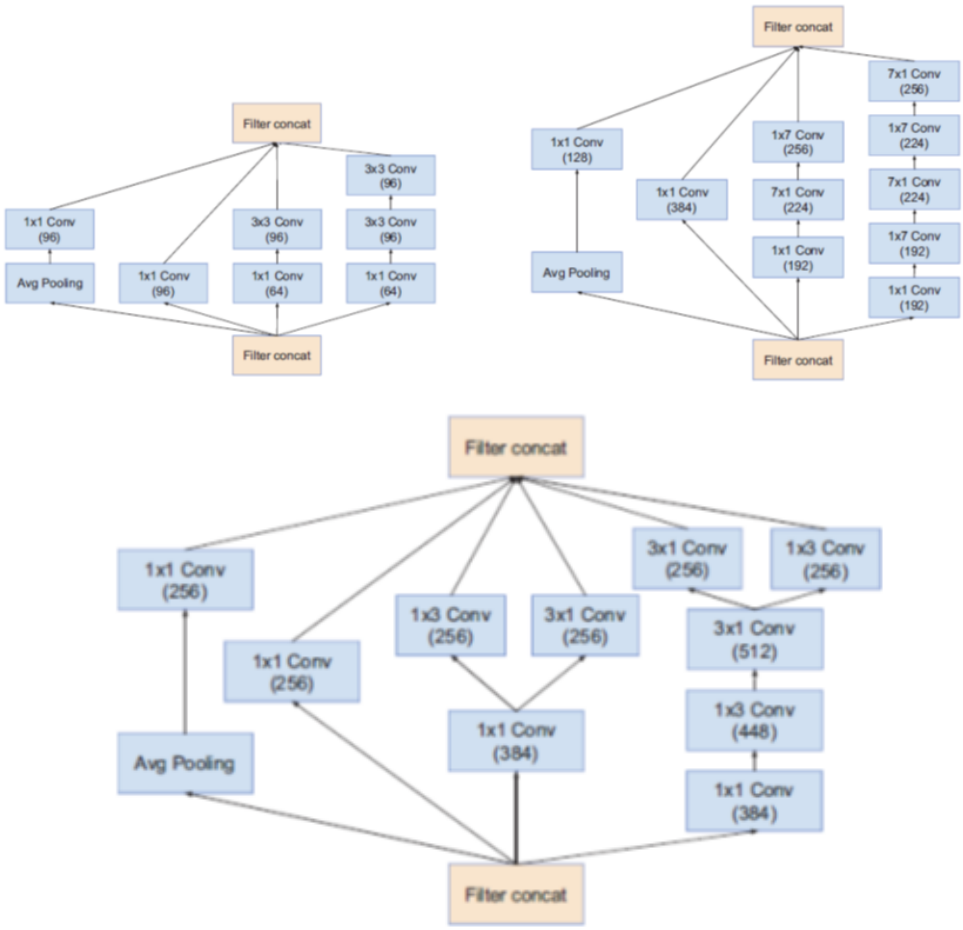


Figura 3.12: A destra la Inception-A, a sinistra la Inception-B, In basso la Inception-C. Fonte ed approfondimenti: [65].

conosciuta ai più come **"aggregated residual transformation"**. Successivamente iniziarono ad essere proposte architetture frutto dell'unione di Inception Layer e Residual Units [69] [70] [71].

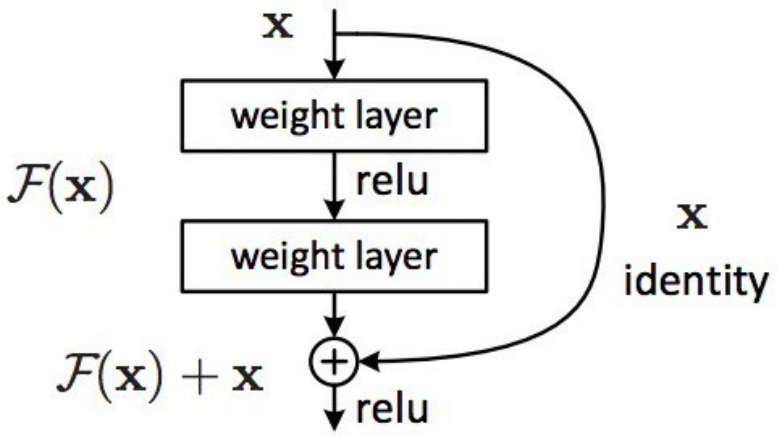


Figura 3.13: Struttura concettuale dei moduli Residuali. Fonte ed approfondimenti: [72].

3.3.10 2016: Inception-ResNet

Grazie all'ispirazione delle ResNet venne progettato un modulo ibrido tra la ResNet e le Inception. Vennero progettate la **Inception-ResNet v1** [52] e la **Inception-ResNet v2**.
La v1 ha dei costi computazionali simili alla Inception v3 mentre la v2 è simile alla Inception v4. Entrambi usano i moduli Inception A, B, C visti nella sezione delle Inception e in figura 3.12 e anche il reduction block. Nella v1 è visibile una connessione "scorciatoia" alla sinistra di ogni modulo, che può essere vista come una dimostrazione del fatto si può andare "in profondità" con l'uso delle reti ResNet.
Nonostante i costi siano similari a quelli della Inception v3, questa ha tempi di allenamento minori pur raggiungendo prestazioni leggermente peggiori in termini di accuratezza. La v2 è molto simile alla precedente ed anche qui si nota il collegamento a sinistra ma la differenza sta principalmente negli iperparametri da settare.

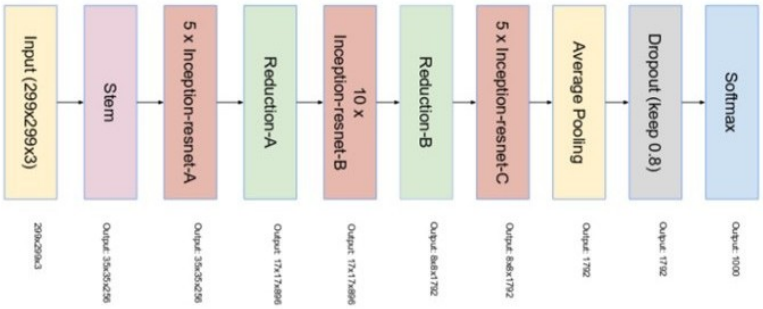


Figura 3.14: Architettura ResNetInception Fonte ed approfondimenti: [68].

3.3.11 2016: PolyNet

PolyNet [73] fu una rete che riuscì a ridurre l'errore ottenuto dalle InceptionResNet v2. Nel 2016 nel ILSVRC fu uno dei progetti di maggiore spicco ma il paper relativo venne esposto nel 2017 quando venne proposto un nuovo modello chiamato PolyInception, una combinazione polinomiale di unità Inception, inserite in percorsi paralleli o a cascata.
Rispetto ai modelli precedenti nei quali le unità erano sequenziali, questa soluzione alternativa permise di aprire a nuove strutture che si differenziassero da quelle convenzionali.
Subito dopo questa venne progettata la **Very Deep PolyNet** che include tre livelli operativi su diverse risoluzioni spaziali. I Moduli PolyInception proposti in figura 3.15 sono i seguenti:

- poly-2: si hanno tre path, uno identità, uno è un blocco Inception singolo e uno è un doppio blocco Inception.
- poly2: il primo blocco Inception può essere condiviso tra i due path seguenti. Ricorda una sorta di Recurrent Neural Network.
- m-poly-2: se il secondo blocco Inception indicato come G non condivide i parametri con F, l'architettura sarà indicata come 1+F+GF.
- 2-Way: è una Polyinception di primo livello, indicata come I+F+G

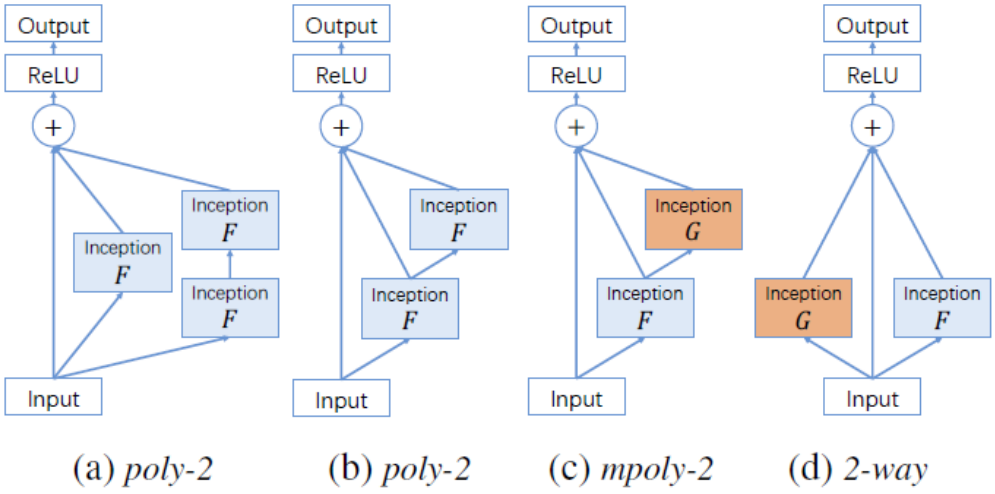


Figura 3.15: I possibili moduli PolyNet. Fonte: [73].

3.3.12 2016: ResNext

ResNext [69] fu una rete che ebbe un grande risalto nel ILSVRC 2016 e in particolare nel contest di classificazione poichè introdusse una nuova dimensionalità relativamente alle reti neurali cioè la **cardinalità**.
In contrasto col concetto di Network in Network questo approccio è detto Network in Neuron e si espande lungo una nuova dimensione. Al posto di una funzione lineare applicata su ogni path qui l'idea è di applicare una funzione non lineare per ogni path.
La cardinalità infatti controlla il numero di trasformazioni complesse, in aggiunta alla profondità e alla grandezza intesa come larghezza.
La ResNext, il cui nome nasce dall'idea della "Next Dimension", va a performare le ResNet, InceptionV3 e la Inception-ResNet-v2, ottenendo ottimi risultati e mostrando un design molto più semplice dei modelli Inception.

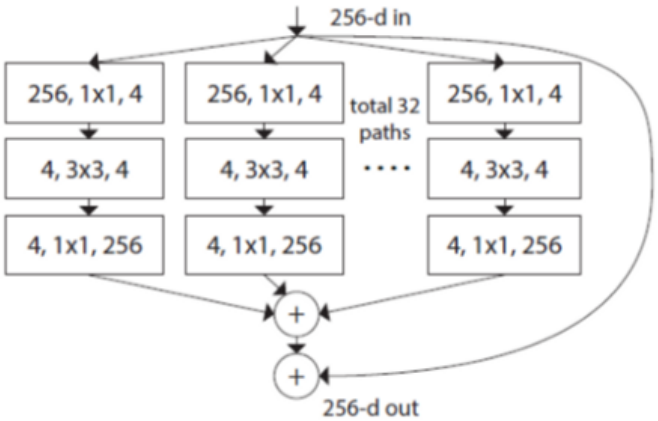


Figura 3.16: Idea base delle ResNext. Fonte: [69].

3.3.13 2016: XCeption

Xception [74] fu un modello architetturale che apportò delle migliorie al modulo Inception e alle relative architetture, proponendo una semplice ma elegante struttura intesa come via di mezzo tra una ResNet e una Inception V4.
L'architettura disponeva di 36 layer convoluzionali, quindi avvicinandosi alla struttura della ResNet34, così come il codice, mentre il modello risultò semplice come la ResNet e più comprensibile rispetto alla Inceptionv4.
Il tipico modulo Inception infatti guarda alla correlazione tra i canali, con una convoluzione 1x1, mappando i dati originali in 3 o 4 spazi, separati e più piccoli dello spazio originale di input e poi mappa le correlazioni in spazi 3D con convoluzioni 3x3 o 3x5.
Xception proposto da Francois Chollet, che fu anche il creatore e sviluppatore principale delle librerie Keras, non è altro che una estensione di Inception che sviluppa tali moduli dando vita a muovi moduli con "convoluzione separabile in profondità". È anche comunemente chiamata **convoluzione separabile**, e consiste in una convoluzione in profondità realizzata in modo indipendentemente su ogni canale di input e seguita da una convoluzione punto per punto. Un'altra differenza riguardò la presenza o assenza di non linearità: Inception contiene la Relu, mentre la convoluzione separabile è implementata senza linearità.

3.3.14 2016: PVANet

I costi computazionali nel nostro contesto ricoprono un ruolo fondamentale, così come le tempistiche di training per cui si cercò di abbassare questi parametri nelle applicazioni pratiche, proponendo nuove strutture che unissero gli elementi dello stato dell'arte alla ricerca dell'accuratezza di predizione.
Nelle **Pvanet** [75] l'idea fu quella di ridurre il numero di canali inserendo più layer e riducendo la ridondanza dei dati grazie alla **C-ReLu (Concatenated Rectified Linear Unit)** [41] [78] e l'Inception Layer. Nello specifico la C-RELU venne applicata negli stati iniziali per ridurre i costi al 50% mentre l'inception Layer venne applicata sulla sottorete per ottenere le features.
Venne inoltre adottata l'idea della rappresentazione multi-scala, che combinò gli output intermedi in modo che i diversi livelli di dettagli e la non linearità potessero essere considerati simultaneamente.

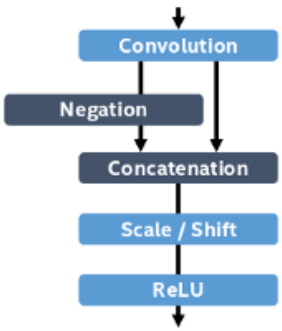


Figura 3.17: La C-Relu può duplicare il numero dei canali di output, andando a concatenare la negazione dell'output prima di applicare la ReLU. Fonte: [75].

3.3.15 2017: DenseNet

GaoHuang e il suo gruppo nel 2017, proposero le **DenseNet** [76] cioè delle CNN "Densely Connected", con dei layer densamente collegati tra loro in modo che l'output del precedente fosse direttamente collegato ai successivi.
Concatenando le feature map apprese da differenti layer si ottiene un aumento di efficienza e aumenta la variazione sugli input: questa è la principale differenza rispetto alle ResNet. Rispetto alle Inception che invece optano anche per la concatenazione, le DenseNet sono più semplici ed efficienti.
Come conseguenza diretta della concatenazione, le feature map sono utilizzabili in tutti i layer successivi alla produzione e viene incoraggiato il riuso all'interno della rete creando un modello sempre più compatto.
Tra gli altri vantaggi riconosciuti è da sottolineare la riduzione della scomparsa del gradiente, la riduzione dei parametri della rete ma anche la qualità di propagazione del flusso di informazioni e del gradiente tra i layer: ogni layer ha accesso diretto ai gradienti, a partire dalla funzione di perdita fino all'input.
Può essere definito uno dei modelli architetturali meglio performanti per il task di classificazione.

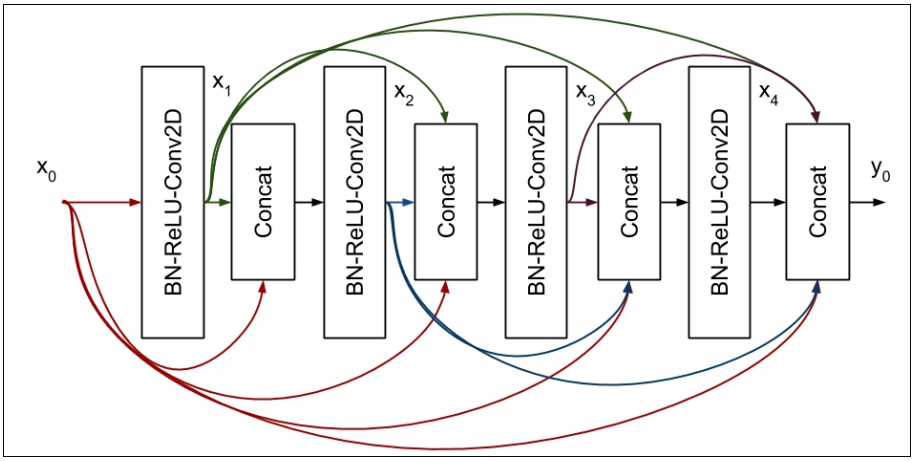


Figura 3.18: Architettura di una DenseNet.

3.3.16 2016/2017: SqueezeNet

La **SqueezeNet** [56] è una delle architetture più recenti e riprende diversi concetti delle ResNet e delle Inception proponendo una migliore architettura a livello di design, dimensionalmente più piccola e con un numero ridotto di parametri. Consente di effettuare un training distribuito in maniera più efficiente, cosa che produce meno overhead. Questi risultano essere i vantaggi di una architettura di dimensioni minori, che presenta una grande accuratezza, paragonabile a quella delle reti AlexNet.
In particolare, per ridurre il numero di parametri e la dimensione si possono usare varie strategie: sostituire i filtri con filtri 1x1 o con filtri 3x3 oppure cercare di diminuire il numero di canali di input o sottocampionare sulla rete in un momento successivo alla convoluzione in modo che il layer convolutivo abbia una mappa di attivazione più grande.
Uno degli elementi, che balza all'occhio, osservando l'architettura è l'assenza di fully connected layer o di dense layer alla fine della cascata di moduli che compongono la rete:

entrambi sono usati per permettere la classificazione e agiscono in modo totalmente differente; in questo caso questa operazione finale è completata all'interno dei moduli **FIRE** della rete.

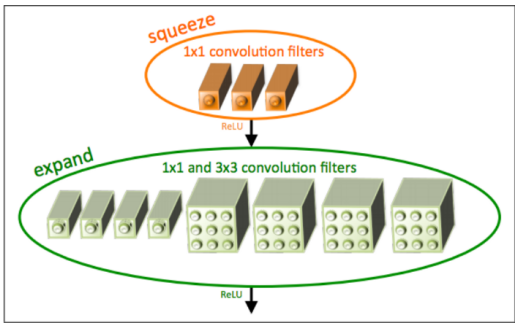


Figura 3.19: Moduli Fire. Fonte ed approfondimento: [77].

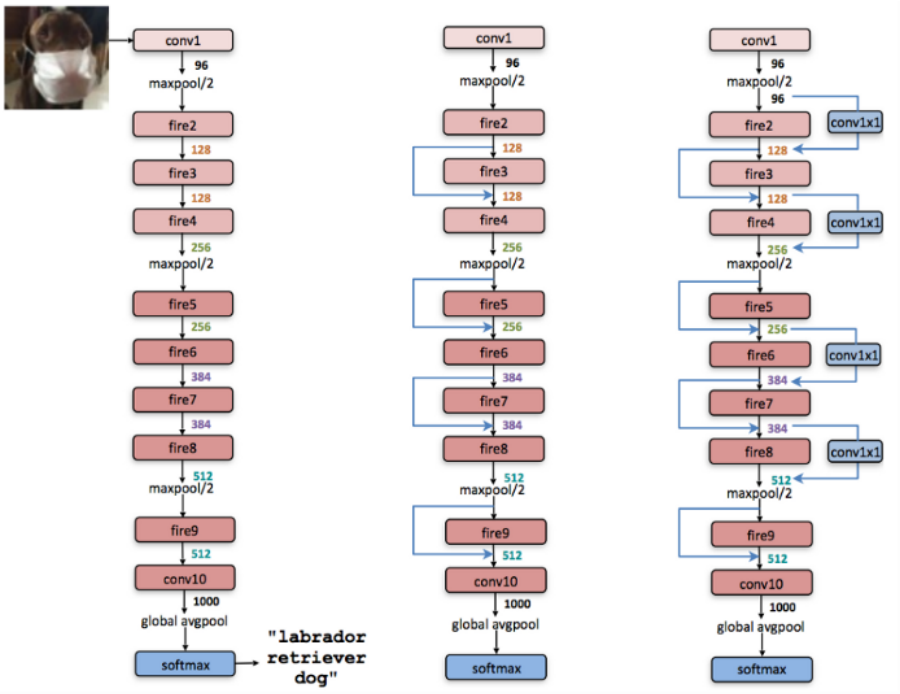


Figura 3.20: Architettura di una SqueezeNet. Fonte ed approfondimento: [77].

3.4 Reti CNN Mobile ed Embedded

È una classe di modelli per applicazioni prettamente mobile o embedded e si tratta di reti caratterizzate da una architettura con **convoluzione separabile in profondità** [74], che permette di creare dei modelli leggeri.
Vengono introdotti due semplici iperparametri globali, per trovare il giusto compromesso tra latenza e accuratezza di previsione e per permettere di creare un modello adattabile al tipo di applicazione per cui è richiesto.
La fattorizzazione, come già detto per le reti Inception, riduce la computazione di calcolo e la dimensione del modello: qui verrà introdotta insieme alla convoluzione separabile che è una fattorizzazione della convoluzione standard in una convoluzione in profondità, seguita da una convoluzione punto per punto.
Questi risultano essere gli approcci per progettare questo tipo reti.
In questa sezione sarà fatta una carrellata di tutte quelle tipologie di reti conosciute ed implementabili su macchine con capacità di calcolo ridotte.

3.4.1 2017: MobileNet

Nelle **MobileNets** [79] la convoluzione punto per punto applica una convoluzione 1x1 per combinare i risultati della precedente convoluzione mentre la convoluzione in profondità invece applica un singolo filtro per ogni canale di input.

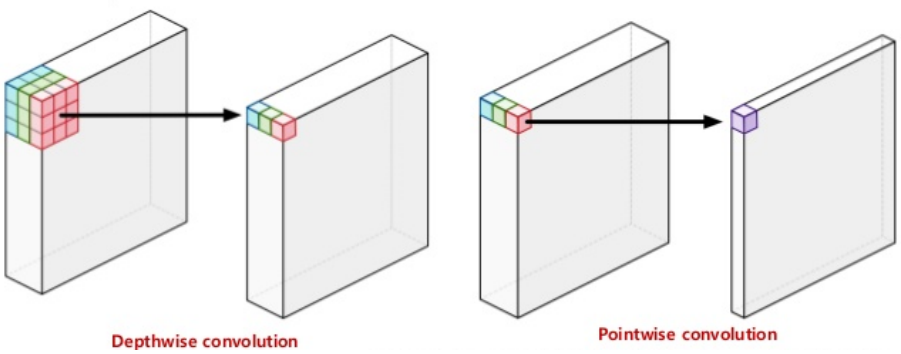


Figura 3.21: A sinistra la Convoluzione Separabile in profondità e a destra la Convoluzione punto per punto. Fonte: [80].

3.4.2 2017: MobilNet V2

La **MobileNet v2** [81] è una delle evoluzioni che migliora lo stato dell'arte delle applicazioni in ambito mobile. Sono state introdotte due nuove caratteristiche: il bottleneck lineare tra i layer e i collegamenti shortcut (o meglio residual connection) tra i bottleneck. I residual block collegano l'inizio e la fine di un blocco di convoluzione con una skip connection e mediante questi nuovi strati si riesce ad accedere ad attivazioni precedenti che non sono state modificate nel blocco di convoluzione: riesco ad avere apprendimento più veloce e migliore precisione.
Le operazioni sono ridotte del 50% rispetto alla versione precedente e sono necessari il 30% in meno di parametri.

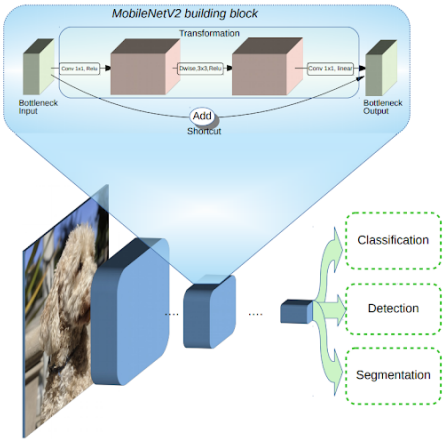


Figura 3.22: Blocco architettuale MobileNet v2. Fonte: [82].

3.4.3 2018: FastDownSampling MobileNet

Le **FD-MobileNet** [83] sono dei modelli in cui il sottocampionamento è completato prima rispetto che in altri modelli mobile in modo da ridurre sensibilmente il costo computazionale e rendere la rete molto efficiente ed accurata, anche in caso di disponibilità di risorse limitata. Nello specifico si applica un sottocampionamento all'interno di 12 layer, la metà rispetto alle precedenti applicazioni MobileNet ottenendo 3 vantaggi: oltre al già citato risparmio computazionale, si incrementano le capacità di immagazzinare informazioni e terzo si ha la possibilità di fare delle valutazioni di immagini in maniera rapida e veloce (Inference).
Dopo il sottocampionamento la rete, si occuperà di completare una sequenza di convoluzioni separabili in profondità.

3.4.4 2017: ShuffleNet

Le **ShuffleNet** [84] [85] sono caratterizzate dalla convoluzione punto per punto di gruppo e dallo shuffle tra canali cioè una sorta di mescolamento mirato a ridurre i costi computazionali e mantenere al contempo una discreta accuratezza.
La ShuffleNet comparata con modelli mobile similari riesce a produrre un buon numero di feature map per immagazzinare più informazioni, cosa che solitamente è problematica per le reti molto piccole ma che qui si riesce a sostenere in maniera molto discreta.
Non è il top tra le applicazioni mobile visto che già le MobileNet sono più accurate, specialmente nelle ultime versioni, però si può definire un buon compromesso costo-efficienza.

3.4.5 2016: SimpleNet

Le reti convoluzionali più conosciute e diffuse come AlexNet, VGGNet, ResNet, GoogLeNet, sono famose per le loro centinaia di milioni di parametri, che le portano ad essere di grandi dimensioni e portano ad una richiesta esagerata di risorse complicandone il relativo training, l'ottimizzazione e l'efficienza.
Di contro le architetture leggere, ottimizzate su questi parametri di valutazione, risultano peccare in termini di accuratezza.
La **SimpleNet** [86] [87] si propone come architettura che può raggiungere un buon trade-off: dispone di 13 layer per svolgere tutto ciò che svolgono le altre architetture divenendo

ottime anche per l'utilizzo su sistemi embedded o mobile.
Come anticipato, dispone di 13 Layer con kernel di convoluzione 3x3 e 2x2 per il pooling mentre gli unici layer che non usano filtri 3x3 sono l'undicesimo e il dodicesimo che usano filtri 1x1.
Viene usata la Batch Normalization prima della ReLU ed è spesso detta l'architettura "No-Dropout".

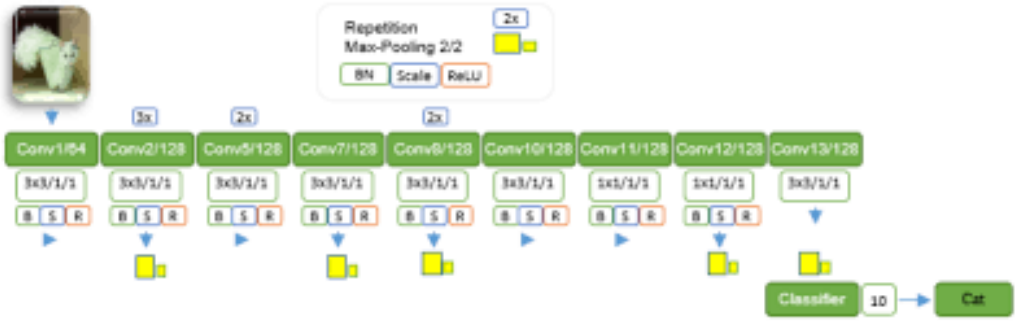


Figura 3.23: Moduli e Architettura SimpleNet. Fonte: [90].

3.4.6 2017: CondenseNet

La **CondenseNet** [88] è un modello convolutivo che combina la "dense connected layers" con un meccanismo di rimozione dei collegamenti non usati. Il primo elemento facilita il riuso delle feature all'interno della rete, visto che la "learned group convolutions" rimuove le connessioni tra layer nei quali il riuso di determinate feature è superfluo.
Le CondenseNet quindi risultano essere più efficienti rispetto alle MobileNet e alle ShuffleNet e sono molto rinomate per gli usi mobile.

3.4.7 Enet (Efficient Neural Network)

Le **Enet** [89] sono il risultato di diverse strategie e risultano essere molto leggere pur offrendo grandi prestazioni. Progettate da Adam Paszke, possono operare anche a livello mobile dal momento che richiedono poca potenza.
Questo algoritmo etichetta ogni pixel nell'image con una classe di oggetti. Viene usato un decoder e un encoder: l'encoder è una CNN usata per la classificazione mente il decoder permette di sovracampionare il risultato dell'encoder.

3.4.8 2017: CapsuleNet

Le **CapsuleNet CNN [91]** sono una classe di modelli usati per riconoscere le features di un oggetto e offrono ottime performance nel riconoscimento oggetti pur avendo dei limiti rispetto alle tradizionali CNN: per esempio il non tener conteggio delle relazioni tra le feature, delle prospettive, delle dimensioni e dei loro orientamenti. Quindi riescono ad esempio, a distinguere bene gli elementi, senza riconoscere il loro posizionamento specifico.
La proposta fu diversa rispetto al passato: Sara Sabour, Nicholas Frost e Geoffrey Hinton pubblicarono un paper chiamato "Dynamic Routing Between Capsule", e poco dopo lo stesso Geoffrey Hinton pubblicò un altro paper che propose dei concetti tratti dal precedente paper ma applicati alle reti neurali. Si iniziò a parlare di Capsule e CapsuleNet.

I risultati proposti furono incoraggianti pur riferendosi alle "digits" cioè elementi disegnati, scritti, tracciati e in particolare venne fatto sia un paragone con le architetture neurali conosciute al momento, sia venne tracciata una nuova idea: il cervello umano possiede dei moduli chiamati "capsule", capaci di rispondere a determinati stimoli visuali in modo ottimale quali riferimenti spaziali come posizione, dimensione, orientamento, velocità.
Il cervello in particolare conosce dei meccanismi di routing, quindi di instradamento delle informazioni di basso livello che permettono di capire quale è la migliore capsula per gestirle nel miglior modo possibile.
Le capsule sono un set di layer neurali innestati ai quali si possono aggiungere altri livelli all'interno degli stessi creando dei contenitori ricorsivi. Il neurone all'interno della capsula cattura le informazioni all'interno di una immagine, e la capsula produce un vettore di output che certifica l'esistenza dell'entità. Tale vettore è poi inviato ai possibili genitori che la capsula individua all'interno della rete neurale; per ciascuno di essi si può trovare un vettore di predizione, che è calcolato basandosi sulla sulla moltiplicazione dei suoi pesi stessi per una matrice di pesi.
Questa, seppur ad alto livello, fu la proposta innovativa delle CapsuleNet.

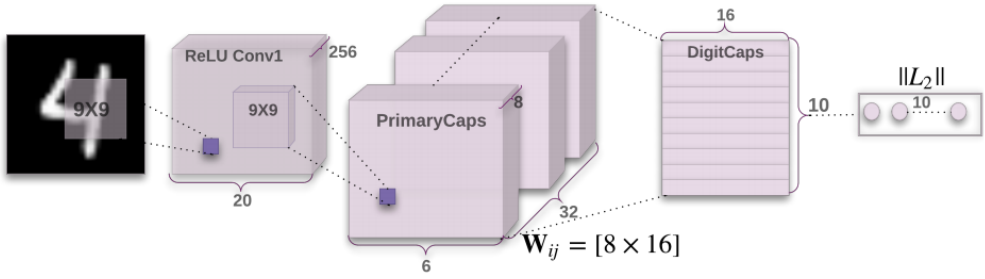


Figura 3.24: Architettura di una CapsuleNet. Fonte: [93].

3.5 Architetture di Rete per Object Detection

Il task di Object Detection può essere diviso in due grosse sottoclassi, analizzate nelle due seguenti sezioni:
Approccio 2-stage: RPN (Region Proposal Network) quindi basato sull'identificazione di alcune regioni spaziali e caratterizzato internamente da due stage operativi.
Approccio 1-: SSD e Yolo che costituiranno la seconda sottofamiglia, e che utilizzano un approccio diverso.

3.6 Region Proposal Network

Questa grande famiglia di reti convoluzionali costituisce la base forte del task appena descritto, inteso come task di classificazione e localizzazione di più oggetti all'interno di una immagine. In questa sezione partendo dalle generiche **Region-CNN** si passerà ad analizzare una serie di miglioramenti implementativi: le **Fast R-CNN**, **le Faster R-CNN** e le **Mask R-CNN**.

3.7 Region CNN

Le **Region CNN** [95] in primo luogo selezionano diverse regioni all'interno di una immagine, per poi passare ad etichettare le categorie di oggetti nelle regioni attraverso una CNN usata per estrarre le features da ogni area proposta.
Nello specifico, queste famiglie di reti sono caratterizzate da 4 passaggi operativi: la **Ricerca Selettiva** che viene effettuata sull'immagine di input e seleziona diverse regioni in alta qualità; poi selezionata una CNN pre-trained, che viene posta prima dell'output, la possibile zona proposta viene trasformata nella dimensione richiesta dalla rete e le feature estrette vengono inviate all'output; subito dopo le feature e le categorie, intese come etichette delle regioni, sono combinate per allenare una vector machine multiple (questa fase è detta **Category Prediction**. Infine, le feature e i box etichettati sono combinati per allenare un modello osservativo a regressione lineare (fase di **Bounding Box Prediction**).
L'alto carico computazionale ha creato molte complicazioni nell'utilizzo visto che è richiesto un tempo eccessivo (quasi 47 secondi) per classificare tutte le regioni di ogni singola immagine e visto che la ricerca selettiva può proporre regioni candidate non ottimali, non basandosi su algoritmi di learning.

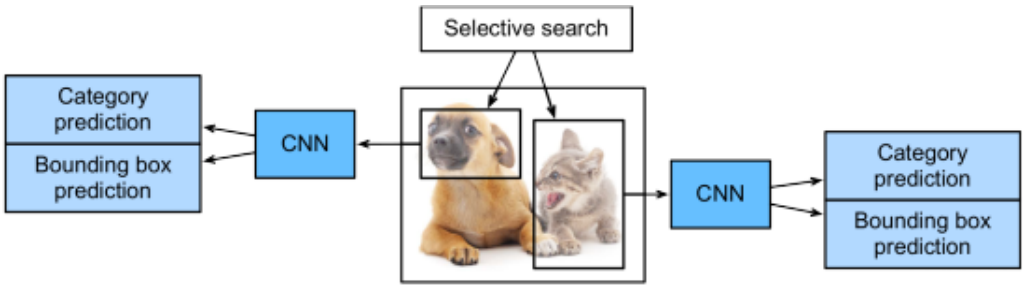


Figura 3.25: Schema esplicativo di una Region CNN based. Fonte ed approfondimenti: [97].

3.7.1 2015-Fast Region Based CNN

Le Fast R-CNN [53] sono un modello migliorativo delle R-CNN: viene usata l'intera immagine come input per l'estrazione delle features piuttosto che tutte le regioni selezionate, come fatto con la ricerca selettiva in precedenza. La CNN seleziona le features dell'immagine generando la tipica feature map e poi da questa sono selezionate le regioni candidate che vengono racchiuse nei box. Da sottolineare l'introduzione del **RoI Pooling Layer** che riceve queste regioni, le rimodella in dimensioni ben definite e le passa ai Layer Fully connected per la classificazione finale.
Risulta lampante il vantaggio rispetto al modello architetturale precedente, visto che con la CNN non sará analizzata una svariata quantità di regioni.

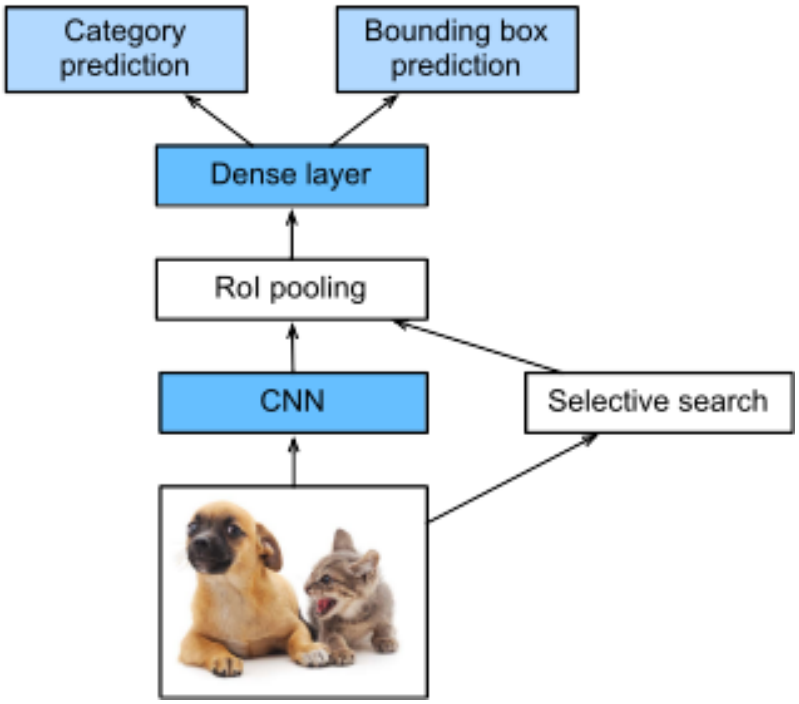


Figura 3.26: Schema esplicativo di una Fast Region CNN. Fonte ed approfondimenti: [97].

3.7.2 2016 Faster R-CNN

Sia la R-CNN che la Fast R-CNN usano la ricerca selettiva per trovare delle regioni, pur sapendo che si tratta di un algoritmo lento e che consuma risorse in maniera eccessiva: questo porta ad un peggioramento delle prestazioni della rete. Shaoqing Ren e il suo team progettarono un algoritmo che permettesse alla rete di individuare le regioni salienti abbandonando l'uso della ricerca selettiva.
Come nella Fast R-CNN anche nelle **Faster** [54] (fig.3.27) l'immagine è inviata alla CNN che provvede a creare una mappa di feature; subito dopo è presente una nuova rete a sé stante, che si occuperà di estarre delle regioni da proporre che saranno poi rimodellate dal RoI Pooling Layer e che verranno usate per classificare l'immagine.

3.7.3 2017: R-FCN

Le **Region-based Fully Convolutional Network** quindi **R-FCN** [96] (fig.3.28) progettate da Microsoft e dalla Tsinghua University propongono delle migliorie nelle tempistiche di Inference pur mantenendo una accuratezza molto elevata.
È considerato come l'approccio più veloce all'interno di questa famiglia di reti per Object Detection che lavorano su due stage. Le RPN tradizionali generano prima le regioni proposte per poi applicare il ROI e passare il tutto ai fully connected layer. Ciò che avviene all'interno degli FC layer dopo il ROI, non implementa nessun tipo di condivisione tra le varie volte che è stato usato il ROI, per cui il processo si allunga.
In questo approccio alla fine i layer FC vengono rimossi e la maggiore complessità è spostata prima dei livelli di pooling ROI per generare le "score maps". Le regioni create con il RPN useranno lo stesso set di score maps usato per creare le predizioni, il tutto all'interno del ROI Layer.
Questa variazione di processo le rende più performanti rispetto alle Faster.

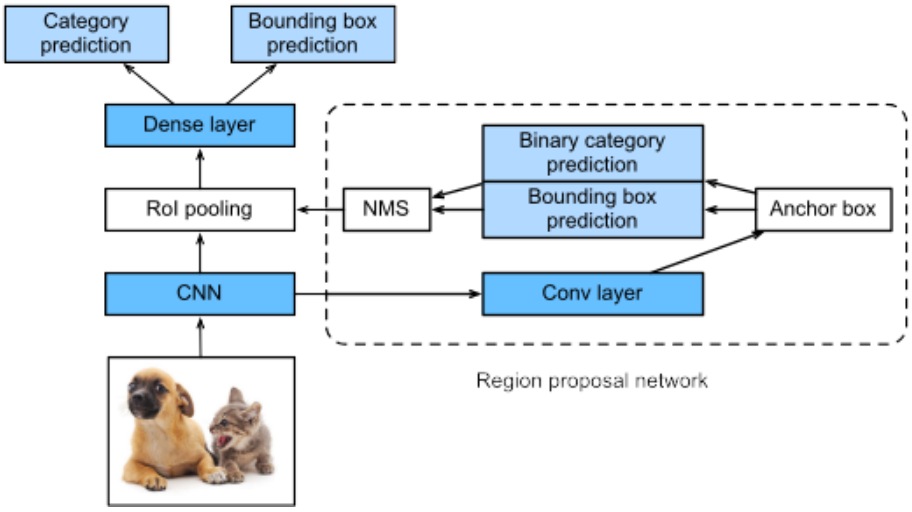


Figura 3.27: Schema esplicativo di una Faster Region CNN. Fonte ed approfondimenti: [97].

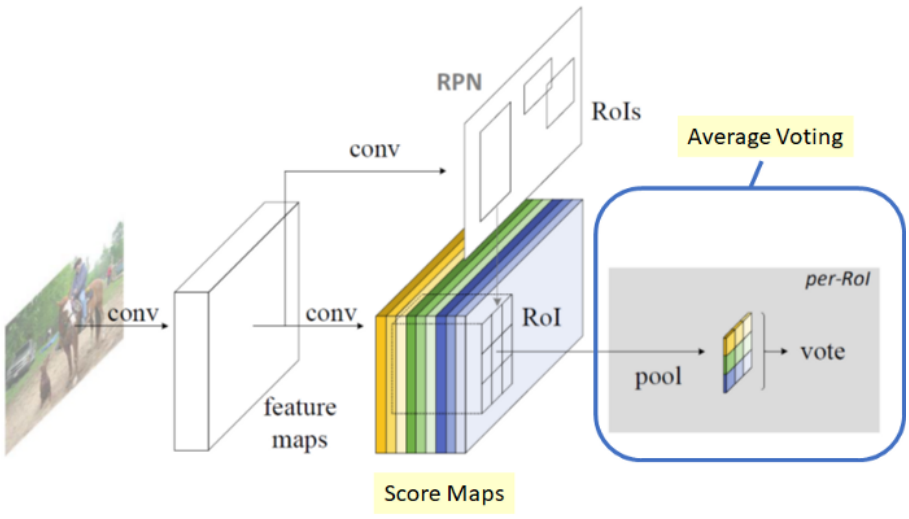


Figura 3.28: Schema esplicativo di una R-FCN. Fonte ed approfondimenti: [98].

3.8 Reti per Semantic Segmentation

La Segmentazione Semantica è una delle tematiche innovative per l'ambito della nostra trattazione.
È una applicazione che sta riscuotendo molto successo dal momento che in questo periodo storico-tecnologico si cercano di ricavare sempre più informazioni dai dati e anche dalle immagini stesse, al fine poi di utilizzare quanto ottenuto in contesti di realtà aumentata, per i veicoli self-drived, per l'interazione macchine-uomo ed altro ancora. Si può anche considerare come un possibile terzo step di applicazione delle reti neurali alle immagini: dopo la classificazione e la localizzazione, ci si può occupare dell'identificazione esatta (o quasi) degli oggetti all'interno dell'immagine. L'applicazione risulta essere molto costosa dal momento che ogni pixel localizzato viene "categorizzato" e perchè si cercherà di delimitare esattamente un oggetto nei suoi contorni.

3.8.1 2016 MASK R-CNN

Le Mask R-CNN [99] sono una variante modificata delle Faster R-CNN in cui il Rol Pooling Layer è sostituito con un Rol Alignment Layer. Il loro scopo però non è solo quello di localizzare ma di segmentare gli oggetti presenti nell'immagine di input.
Questa struttura permette di usare l'interpolazione bilineare per ottenere informazioni spaziali sulle feature map, rendendo la Mask R-CNN migliore per la predizione pixel per pixel. Poi per il resto il Rol restituisce in output lo stesso risultato delle architetture precedenti ma ci permette, se volessimo, di aggiungere una nuova rete CNN per predire la posizione dell'oggetto a livello di singolo pixel.

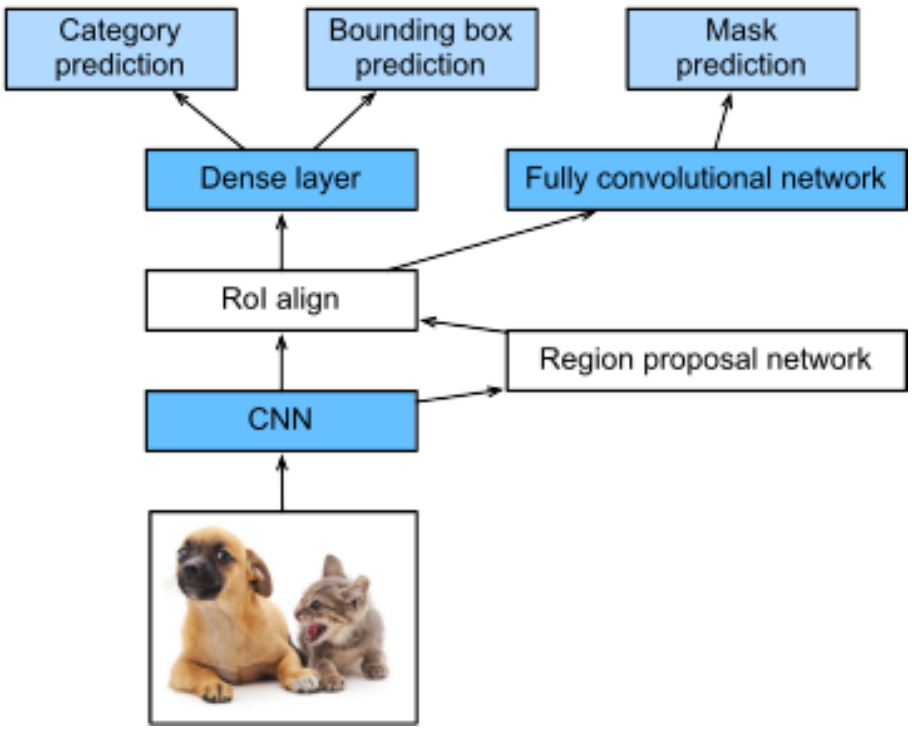


Figura 3.29: Schema esplicativo di una MASK R-CNN. Fonte ed approfondimenti: [97].

3.8.2 2016: ParseNet

ParseNet [100] è una architettura neurale, specializzata per la segmentazione semantica. L'approccio è molto semplice e sfrutta le average feature per un layer al fine di aumentare le feature specifiche di ogni locazione.
Riguardo a questa rete non vi sono grandissime spiegazioni a livello tecnico sulla struttura quindi si farà semplicemente una presentazione ad alto livello.

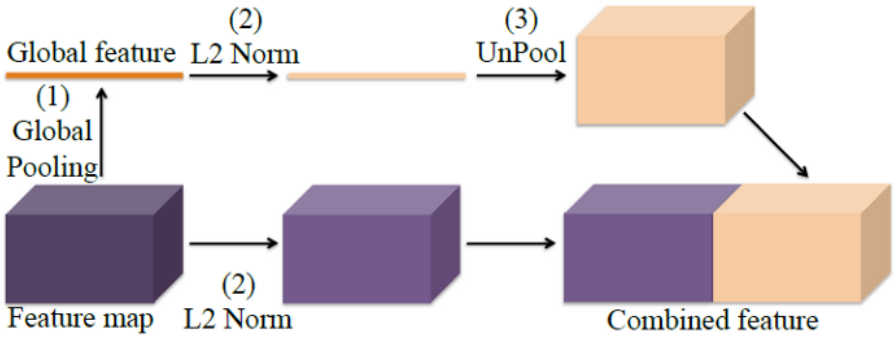


Figura 3.30: Schema esplicativo di una ParseNet. Fonte ed approfondimenti: [101].

3.9 Un Differente tipo di approccio

Fortunatamente negli ultimi anni sono state create nuove architettura per ovviare al problema del collo di bottiglia delle R-CNN. In questa sezione sono inserite le applicazioni di Real-Time Object Detection: **Yolo** and **SSD** sono le più conosciute e le più diffuse in termini di utilizzo.
Si tratta della seconda famiglia di approcci in merito al task di Object Detection e in particolare quella dell'approccio one-stage.

3.9.1 2016 YOLO: Real-Time Object Detection

You only look once (YOLO) è [102] [103] delle architetture stato dell'arte per l'Object Detection.
Finora i sistemi visti per l'Object Detection hanno sempre riproposto classificatori e localizzatori per riconoscere un oggetto: questo approccio invece prevede l'uso di una singola rete. L'immagine verrà divisa in regioni e verrano predetti i box relativi e la probabilità per ogni regione. Il meccanismo di base appare semplice: l'immagine di input viene divisa in una griglia S x S e se il centro di un oggetto finisce all'interno di una cella, quella cella è responsabile del riconoscimento di quel determinato oggetto. Ogni cella predice un tot di box e produce un punteggio di confidenza che corrisponde a quanto il modello è corretto in termini di posizionamento e predizione.
La pipeline quindi è singola ed è ottimizzata per una singola operazione, e questo porta ad estremizzare positivamente le prestazioni: risulta essere una architettura velocissima che processa real time circa 45 frames per secondo, mentre la versione ridotta FAST YOLO riesce a processare 155 frames per secondo.
Unica pecca negativa riguarda il fatto che Yolo rispetto alle altre reti commette più errori nella predizione fornendo falsi positivi magari sugli sfondi delle foto.

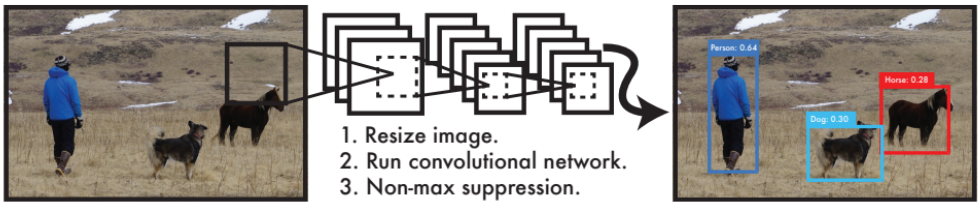


Figura 3.31: Funzionalità di base di Yolo. Fonte: [102].

L'immagine viene ridimensionata in 448x448, viene lanciata una singola CNN e viene ottenuto il risultato. L'architettura si ispira alla GoogLeNet ma a differenza di quella al posto del modulo Inception viene usato un layer di riduzione 1x1, seguito da moduli convoluzionali 3x3: la rete risulta avere 24 layer convolutivi seguiti da 2 layer fully connected mentre la versione FAST consta di un minor numero di layer convolutivi cioè 9.
La versione 1 è stata seguita da **Yolo v2 [104]**, la quale ha portato dei miglioramenti mediante l'inserimento di alcune modifiche architetturali, visto che Yolo rispetto alle Faster R-CNN mostrava un numero maggiore di errori di localizzazione: è'stata introdotta la Batch Normalization per regolarizzare il modello, è stata migliorata la classificazione alla piena risoluzione 448x448 ma soprattutto risultano importanti le **Anchor Boxes**. Yolo prediceva le coordinate dei bounding box usando direttamente i fully connected layer invece con queste Anchor Boxes predicendo gli offset al posto delle coordinate si semplifica il problema e l'apprendimento diventa più semplice.

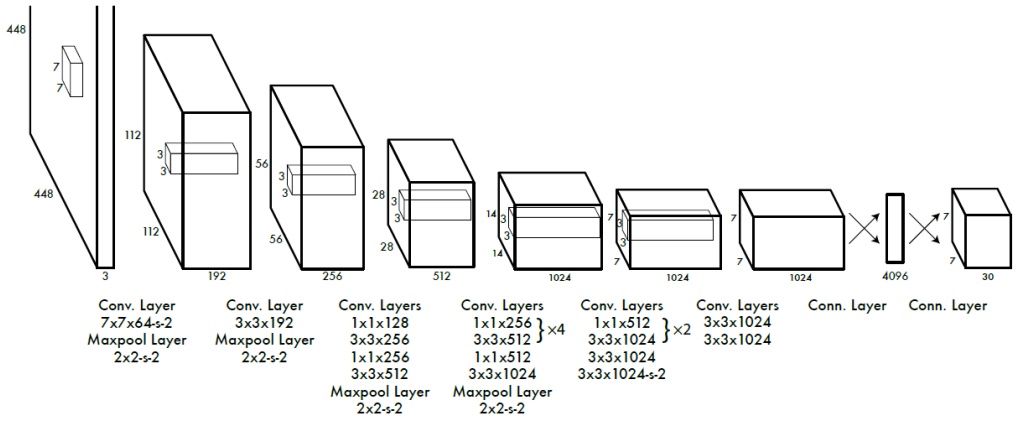


Figura 3.32: Yolo v1. Fonte: [102].

La prima versione riusciva a predire solamente 98 boxes per immagine mentre nella successiva più di un centinaio. La V2 inoltre predice su una mappa 13x13, aggiungendo un layer che funge da filtro passa-basso, portando le features dal vecchio layer 26x26 al nuovo: quindi si passa da 26x26x512 feature a 13x13x2048.
Yolo v2 è stato chiamato Yolo9000 per la capacità di predire oltre 9000 categorie.
YOLOv3 [105] fa predizioni su 3 diverse scale, riducendo l'immagine rispettivamente in scala 32, 16 e 8, allo scopo di rimanere accurata anche su scale più piccole (le versioni precedenti avevano dei problemi con le immagini piccole).
Per ciascuna delle 3 scale, ogni cella è responsabile della predizione di 3 bounding box, utilizzando 3 anchor boxes. Inoltre, risulta molto più precisa delle versioni precedenti, e pur essendo un po' più lento, rimane comunque uno degli algoritmi più veloci in circolazione. La v3 usa come architettura una variante della Darknet, con 106 layer convoluti.
Interessante è anche **Tiny YOLO**, funzionante su **Tiny Darknet**, e in grado di girare su dispositivi limitati come gli smartphone.

DarkNet

Yolo si poggia quindi su DarkNet, una rete neurale di cui sono stati progettati vari modelli ma dei quali non vi è molta descrizione sui paper scientifici: fondamentalmente YOLO viene considerato una sorta di framework scritto in C e che può sfruttare il supporto CPU e GPU.
Ad esempio, per Yolov2 è stato usata la DarkNet-19, molto simile alle VGG e con filtri 3x3 e un numero doppio di canali dopo ogni layer di pooling. Su questo modello, sulla base delle intuizioni delle NIN è stato usato un "global average pooling" per completare le predizioni, piuttosto che usare filtri 1x1. Questo modello ha 19 layer convolutivi e 5 max-pooling layer. Per l'Object Detection sono stati aggiunti poi 11 layer.
Yolo v3 invece, usa una variante con 53 layer, con 53 layer aggiuntivi per classificazione che portano il computo a 106 layer totali. Questo la rende chiaramente più pesante.

Type	Filters	Size/Stride	Output
Convolutional	32	3 × 3	224 × 224
Maxpool		2 × 2/2	112 × 112
Convolutional	64	3 × 3	112 × 112
Maxpool		2 × 2/2	56 × 56
Convolutional	128	3 × 3	56 × 56
Convolutional	64	1 × 1	56 × 56
Convolutional	128	3 × 3	56 × 56
Maxpool		2 × 2/2	28 × 28
Convolutional	256	3 × 3	28 × 28
Convolutional	128	1 × 1	28 × 28
Convolutional	256	3 × 3	28 × 28
Maxpool		2 × 2/2	14 × 14
Convolutional	512	3 × 3	14 × 14
Convolutional	256	1 × 1	14 × 14
Convolutional	512	3 × 3	14 × 14
Convolutional	256	1 × 1	14 × 14
Convolutional	512	3 × 3	14 × 14
Maxpool		2 × 2/2	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	512	1 × 1	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	512	1 × 1	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	1000	1 × 1	7 × 7
Avgpool		Global	1000
Softmax			

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figura 3.33: DarkNet-19 a destra, DarkNet53 a sinistra. Fonte: Sito Ufficiale [103].

3.9.2 2016: SSD MultiBox

Questa architettura è stata rilasciata intorno alla fine del 2016 e raggiunse ottimi livelli in termini di prestazioni.
SSD [106] è una sigla che indica Single Shot Detector anche se il vero nome è Single Shot Multibox Detector: single shot poiché l'operazione viene svolta in un singolo passo dalla rete mentre il multibox è la tecnica utilizzata per riconoscere gli oggetti.
SSD velocizza il processo eliminando l'uso della regione proposta ma per recuperare la perdita di accuratezza usa il multi-scale features e i default boxes: tutto ciò la rende accurata come le Faster R-CNN ma anche più veloce sia rispetto alla Faster sia rispetto alle prime versioni di Yolo.
L'architettura di base è quella della VGG-16 senza i fully connected layer, scelta poichè era una delle più conosciute ed usate e perchè permetteva di fare molto Transfer Learning. I fully connected layer sono stati sostituiti con un set di layer convoluzionali aggiuntivi. I blocchi veri e propri usati per l'Object Detection sono due: estrazione della feature e applicazione dei filtri convolutivi per riconoscere gli oggetti. Le features sono estratte con struttura VGG16 con convoluzione 4x3 (ad esempio) quindi poi verranno applicati dei piccoli filtri convolutivi per l'Object Detection. Vengono ottenuti dei feature layer di dimensione m x n con p canali (m x n x p) e viene applicata una convoluzione 3x3 e per ogni locazione sono ottenuti k box di dimensioni differenti; per ciascuno dei k box vi sono c punteggi di classe e 4 offset relativamente alla forma originale del box. Si otterranno così (c+4)x k x m x n output. La famiglia di reti appena esposta dispone di due modelli:

- SSD300: lavora su immagini 300x300 di risoluzioni ridotte in input ed è una struttura molto veloce.
- SSD512: lavora su immagini 512x512 in input di risoluzione elevata e propone risultati molto accurati.

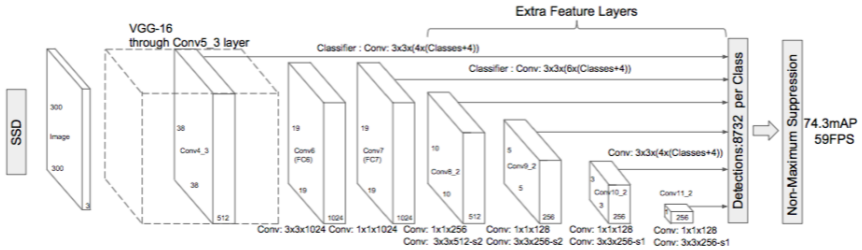


Figura 3.34: SDD Multibox: Architettura. Fonte ed approfondimenti: [107].

3.9.3 2017: RetinaNet

L'architettura **RetinaNet** [108], proposta dal FAIR, Facebook AI Research, è principalmente utilizzata per estrazione delle feature.
È una singola rete complessa composta da un backbone e due specifiche sottoreti.
Il backbone è responsabile della creazione di una nuova mappa convolutiva di feature a partire dall'intera immagine mentre la prima subnet completa la classificazione a partire dall'output del backbone; la seconda rete si occupa invece dei bounding box e della loro identificazione.
Il backbone, nello specifico, è composto da una rete che può essere una ResNet50 o 101 seguita da una **Feature Pyramid Network** [110] che permette di costruire una piramide di feature multiscala a partire dall'immagine.

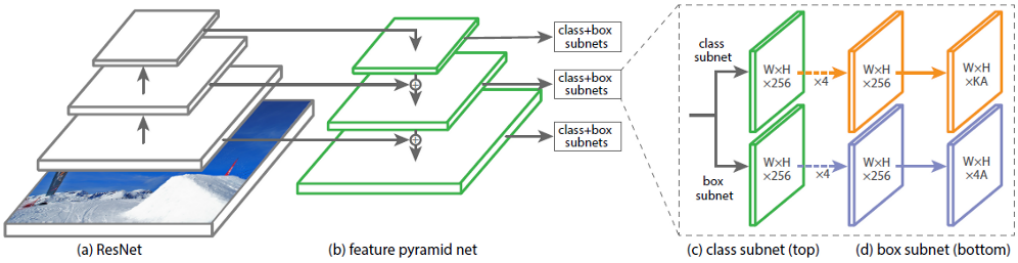


Figura 3.35: Architettura RetinaNet. Fonte ed approfondimenti: [109].

3.10 Video Classification

I risultati ottenuti nell'ambito della classificazione delle immagini hanno incoraggiato delle valutazioni in merito alle CNN e alla loro possibile applicazione nell'ambito dei video, nei quali le reti, non hanno accesso solo alle informazioni presenti in una sola immagine statica ma anche ad una complessa evoluzione temporale.
Ovviamente si va incontro ad un problema molto più complicato già a partire dai dataset visto che i video sono molto più complicati da catalogare annotare e salvare anche per via delle dimensioni maggiori; in secondo luogo bisognava andare a vedere quale pattern di rete fosse il migliore per apprendere informazioni locali più dettagliate in modo da influenzare in maniera più consona le predizioni, e soprattutto andare a valutare le tempistiche di allenamento delle reti, in modo tale da ottimizzare il numero di parametri e ridurre in maniera ottimale i tempi di allenamento.
I primi studi [111] per avvicinarsi al problema furono portati avanti usando un dataset di 1 milione di video sportivi tratti da Youtube divisi in 487 categorie e poi dal momento che il problema si spostò dall'analisi della singola immagine all'analisi di un flusso di immagini continue e ravvicinate cioè i vari frame che compongono il video, l'idea per migliorare l'approccio fu quella di dividere lo stream di processamento in due parti: una che impara le features a bassa risoluzione, l'altra che apprende ad alta risoluzione e che opera sulla porzione centrale del frame. Questo approccio ha ridotto la dimensionalità dell'input.
Con i feature appresi durante l'utilizzo di queste tecniche sul dataset citato, si è provato a fare una sorta di transfer learning su un dataset minore cioè UCF-101, che consta di 13.320 video di 101 categorie: si notarono dei miglioramenti rispetto al fare direttamente allenamento su un dataset più piccolo come l'UCF.
Oltre al già citato problema delle dimensioni del dataset sorse un nuovo problema: la lunghezza variabile di un video rispetto ad un altro. Si cercò di lavorare almeno inizialmente su spezzoni/ritagli di video cercando poi di aggregare le predizioni tra i vari spezzoni fino a giungere grazie a Karpathy e al suo gruppo alla definizione di 3 approcci diversi per classificare un frame alla volta:

- Single Frame: si aggregano le predizioni ottenute da più frame/immagini.
- Late Fusion: vengono combinati/concatenati il primo e l'ultimo frame.
- Early Fusion: viene preso un più lungo e contiguo segmento per il video.
- Slow Fusion: il più complicato, nel quale vengono sovrapposti parzialmente 4 segmenti contigui nei layer convolutivi.

Il metodo sperimentato che ebbe più successo fu lo Slow Fusion, seppur di poco rispetto al Single Frame che risultò essere quello più semplice: in generale il miglior risultato si ottenne combinando i metodi in un altro metodo combinato: **Single +Early + Late + Slow**.

Venne anche sviluppato il modello di CNN multirisoluzione andando a separare l'input in due stream: l'input veniva passato a due livelli convolutivi separati che però si ricongiungevano solo dopo due sequenze isolate di Convoluzione + MaxPooling + Batch Normalization. Questo fornì una forte riduzione del tempo di Convoluzione.
In termini molto spiccioli, l'idea più plausibile visti i risultati ottenuti sembrò quella di combinare le informazioni riguardo alle immagini che temporalmente compongono un video. Sfruttare le architetture convoluzionali parve qualcosa di naturale, visto che queste fornivano grandi prestazioni nell'Object Detection ma visto anche che valutare il singolo

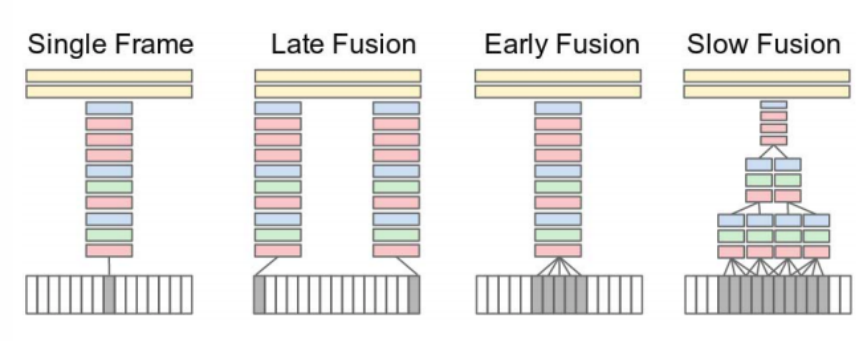


Figura 3.36: Metodi di combinazione. Fonte ed approfondimenti: [37].

frame e fare una media non forniva informazioni complete per una accurata classificazione dei video.
Necessitava però con questa strategia una grande quantità di frame per ottenere una precisione maggiore.
Quindi se da un lato si cercò di usare strutture convoluzionali già conosciute che facessero uso di tecniche di pooling adattate, dall'altro si fece parallelamente strada l'idea dell'uso di **Reti Neurali Ricorrenti** [112]. Ogni frame era processato indipendentemente con le tecniche di pooling usando una CNN, mentre la Rete Neurale Ricorrente usata era derivata dal modello **LSTM (Long Short Term Memory)** che venne sfruttato per acquisire informazioni a livello temporale.
Condividendo i parametri nel tempo, entrambe le architetture sono capaci di mantenere un numero costante di parametri mentre catturano delle informazioni globali sull'evoluzione temporale del video. Questo tipo di approcciò era molto dispendioso a livello computazionale dal momento che faceva uso di diversi filtri convoluzionali 3D, allora fu necessario ipotizzare qualcosa che abbattesse questa problematica: processare un frame al secondo. Questa scelta faceva perdere le "motion information" cioè le informazioni sugli spostamenti ma queste potevano essere compensate inserendole in forma di flusso ottico (flusso ottenuto dall'analisi di frame adiacenti). Questa seconda strategia fu quella su cui cadde la scelta.
Le architetture CNN usate furono due: AlexNet e GoogLeNet, architetture già analizzate in precedenza: in questa sezione in aggiunta saranno analizzati anche i modelli di Pooling specifici usati in ambito video, evitando di trattare in maniera dettagliata ciò che riguarda le Reti Neurali Ricorrenti. La prima strada di analisi, inerente alla combinazione delle informazioni tra più frame si incentrò sulla ricerca di una struttura di layer di Max-Pooling che abbinasse prestazioni e risultati ottimali: questo layer infatti dimostrò di essere performante in termini di apprendimento veloce dal momento che l'aggiornamento del gradiente è generato da un set di features sparso relativo a tutti i frame.
Le architetture di Max-Pooling analizzate furono le seguenti:

- ConvPooling: il max pooling è fatto sul layer di convoluzione finale sui frame del video, in modo che le informazioni spaziali nell'output siano preservate nel dominio del tempo.

- Late Pooling: prima le feature passano su 2 layer fully connected e poi dal max pooling, e i pesi dei layer convolutivi e dei layer fully sono condivisi. Questa implementazione combina direttamente informazioni di alto livello tra i frame.

- Slow Pooling: combina le informazioni dei frame dentro finestre temporali; il max pooling è applicato sulle feature di 10 frame per poi passarlo ad un fully connected layer con pesi condivisi mentre nella seconda parte viene combinato l'output di un singolo max pooling layer con l'output di tutti i fully connected layer. Vengono così combinate le informazioni locali prima di combinare quelle di più frame temporalmente sparsi.
- Local Pooling: simile allo Slow, sono combinate le feature locali dopo l'ultimo layer convolutiva. Ho un solo layer di pooling e non rischio di perdere informazioni temporali. Ovviamente subito dopo ho due fully connected layer con pesi condivisi e un ampio soft-max layer che combina tutto.
- Time-Domain Convolution: contiene un layer convolutivo temporale aggiuntivo prima del layer di pooling in modo che il Max Pooling sia fatto su un dominio anche temporale: è composto da 256 kernel di dimensione 3x3 che lavorano lungo 10 frame, 5 frame alla volta (5 stride frames). Riesco a catturare informazioni in una piccola finestra temporale.

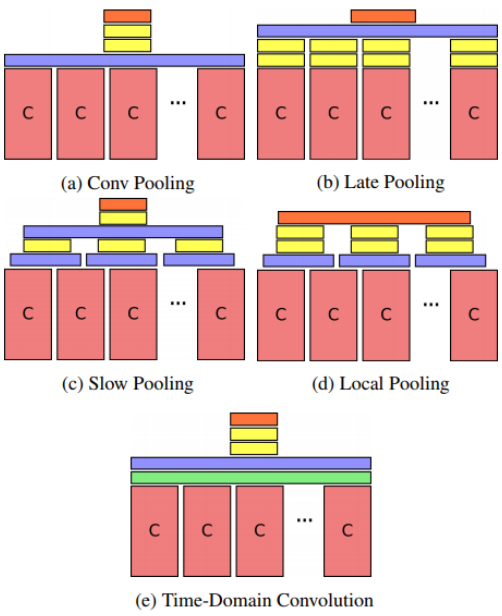


Figura 3.37: Architettura di Pooling. Fonte: [112].

Usando questi modelli con architettura AlexNet sul Dataset Sports-1M si evidenziò che il Late Pooling era quello con risultati peggiori e che il Time-Domain Convolution dava risultati meno completi rispetto agli altri modelli: il ConvPooling era il modello migliore. Ma il risultato più importante che venne fuori è che era necessario studiare un nuovo modello più sofisticato che usasse per l'apprendimento sia più domini temporali che un singolo dominio temporale: le Reti Ricorrenti insieme all'uso delle cnn quindi risultarono essere il modello più ovvio in ambito video per la classificazione di oggetti/soggetti tra i frame.

Capitolo 4

Lo Stato Dell'Arte degli applicativi: Framework, Dataset, API,

4.1 Introduzione allo Stato dell'Arte Applicativo

In questo capitolo verrà discusso tutto ciò che realmente ci permetterà di mettere in pratica le nozioni teoriche viste nella parte relativa alle reti neurali. Partendo dai Dataset che sono l'input da fornire alle nostre reti si finirà con l'analizzare i framework, cioè quei programmi o applicativi che permettono di lavorare sulle reti Neurali. Successivamente saranno passate in rassegna anche alcune Api che operano sopra alcuni framework ma anche dei progetti di Api/software a se stanti, che poggiano su framework o che operano in maniera del tutto indipendente quali Detectron e DetecTorch.

4.2 Dataset

4.2.1 Cenni introduttivi sui dataset

Uno degli elementi fondamentali per il machine learning è la disponibilità di dataset cioè delle collezioni di dati adeguati (in formato diverso in base al task di pertinenza) da utilizzare per le fasi di training e testing, oltre che di valutazione finale del modello.
Chiaramente saranno dettagliati dataset che contengono delle immagini e in particolare per ogni categoria presente nel dataset, dovranno essere presenti un ampio numero di immagini cercando di non avere un numero di elementi troppo diverso fra tutte le classi, al fine di evitare situazioni di sbilanciamento.
Per rendere la rete più robusta sarà necessario disporre di immagini che rappresentino l'oggetto da classificare con angolazioni diverse, con sfondi diversi e magari anche con situazioni di luminosità differenti.
Per ciascuna categoria del dataset, le immagini raccolte dovranno essere suddivise in un training set ed un validation set che contiene una quantità ridotta di immagini.
È importante tuttavia fare attenzione al fatto che le immagini presenti nel validation set, non siano presenti anche nel training set in modo da non rendere falso il valore di accuratezza. Per ogni classe di immagine, la situazione ottimale sarebbe quella di avere nel training set sia immagini rappresentanti l'oggetto in questione con uno sfondo e sia

immagini rappresentanti solamente l'oggetto senza nessuno sfondo: in questo modo la rete durante la fase di allenamento diventerebbe più robusta, riuscendo a distinguere in maniera migliore l'oggetto di interesse anche in situazioni più complicate. Spesso si tende a fare uso di un dataset già disponibile e reperibile online anche per il download sul sito ufficiale del progetto, seppur questi possano raggiungere dimensioni considerevoli.
In questa sezione quindi sarà fatta una carrellata sui dataset più conosciuti e più utilizzati, sottolineando le peculiarità e le caratteristiche di ciascuno, oltre ad alcuni cenni storici.

4.2.2 ImageNet

https://wordnet.princeton.edu/
http://www.image-net.org/about-overview

Imagenet è stato ampiamente descritto e discusso a livello storico nella sezione introduttiva sulle reti neurali, in quanto il relativo contest fu il punto di partenza per esplorare il contesto centrale della nostra trattazione. Nello specifico il dataset di ImageNet [113], è un database organizzato sulla base della gerarchia **WordNet**, nella quale ogni concetto è descritto da più frasi e parole: potremmo considerare un concetto come set di sinominimi o synset; seguendo la stessa logica ogni nodo è raffigurato da centinaia/migliaia di immagini.
Al momento esistono in media oltre 500 immagini per nodo e per ciascuno di essi ImageNet compila una lista accurata di immagini dal web.

4.2.3 PASCALVOC DATASET

http://host.robots.ox.ac.uk/pascal/VOC/
http://host.robots.ox.ac.uk/pascal/VOC/databases.html

Il progetto **Pascal Voc** [114] fornisce un dataset per il riconoscimento oggetti e un set di strumenti per accedere al set e alle annotazioni.
A partire dal 2005 fino al 2012 venne portata avanti una sfida di valutazione delle prestazioni che incoraggiò due tipi di testing dei metodi: metodi che sono allenati solo usando il set di training e valutazione (trainval) e metodi che sono testati usando ogni tipo di dato eccetto i test set.
La valutazione dei nuovi metodi sui dataset forniti può essere reperita attraverso il Server di Valutazione che viene reso disponibile sul sito ufficiale, insieme al Dataset vero e proprio. Le immagini sono reperite in formato PNG dal sito "Flickr" seguendo la legislazione relativa al sito di pertinenza in merito al loro sfruttamento.

4.2.4 SUN Database

https://groups.csail.mit.edu/vision/SUN/
https://vision.princeton.edu/projects/2010/SUN/

Lo scopo del **Dataset Sun** [115] [116] è quello di fornire ai ricercatori una ampia quantità di immagini annotate che coprono una larga varietà di scene ambientali, luoghi e oggetti correlati a questi.
È stato usato il dizionario WordNet come in Imagenet, in modo da stabilire un dizionario per le scene specifiche e poi successivamente collezionare le relative immagini mediante strumenti di ricerca e alla fine annotare ciascuna di esse manualmente.
SUN, il cui acronimo significa Scene Understanding, contiene 899 categorie e 130.519 immagini e usa 397 categorie per valutare gli algoritmi e le relative prestazioni in ambito "scene recognition".

4.2.5 SIFT10M Dataset

http://archive.ics.uci.edu/ml/datasets/SIFT10M

Il **Dataset SIFT10M** [117] è usato principalmente per valutare il metodo di approssimazione al vicino più prossimo, e il titolo dei file, in formato PNG, indica la colonna in cui è posizionato il feature Sift.
Ogni punto è una feature sift estratta da Caltech-256 grazie alla libreria open source VLFeat.

4.2.6 CALTECH-256

http://www.vision.caltech.edu/Image_Datasets/Caltech256/

Caltech-256 [118] è la nuova versione del dataset Caltech-1, le cui immagini sono ottenute da Google and PicSearch. Il 32% del totale delle immagini è valutata come good seguendo una scala che comprende 3 parametri: good, bad con foto confuse, artistiche o poco definite e not applicable per le categorie non presenti.

Dataset	*Released*	*Categories*	*Pictures*	*Pictures Per Category*			
			Total	*Min*	*Med*	*Mean*	*Max*
Caltech-101	2003	102	9144	31	59	90	800
Caltech-256	2006	257	30608	80	100	119	827

4.2.7 COCO Dataset

http://cocodataset.org

Coco(Microsoft Common Objects in Context) [119] è un ampio dataset per il riconoscimento di oggetti e soggetti umani e per la relativa classificazione ed entra proprio nel cuore della nostra ricerca; in particolare, gli oggetti e la loro posizione all'interno di queste (le immagini reperibili per una determinata categoria non contengono solo il singolo oggetto ma contengono un contesto ben determinato in cui sono riconoscibili altri elementi): questo portò nel tempo ad avere la necessità che ogni possibile oggetto fosse categorizzato e quindi di conseguenza la necessità di avere un database su larga scala.
La ricerca della soluzione per questo problema portò a questo dataset: COCO contiene 91 categorie di oggetti comuni, di cui 82 ne contengono almeno 5000 etichettate, per un totale di circa 330.000 immagini di cui oltre 200mila già etichettate, per un totale di 2.5Milioni di oggetti categorizzati. Quindi risulta essere di grande aiuto nell'ambito della localizzazione oggetti nelle immagini in due dimensioni, essendo molto più grande dei dataset altrettanto conosciuti come PASCAL VOC e SUN e IMAGENET.

4.2.8 Google OpenImages Dataset

https://storage.googleapis.com/openimages/web/index.html
https://github.com/openimages/dataset

Open Images V4 [120] è un dataset di 9.2M di immagini, con annotazioni unificate per la classificazione, riconoscimento di oggetti e riconoscimento di relazioni visuali nelle immagini stesse.
Le immagini seguono la Creative Commons Attribute License che lo contraddistingue rispetto ad altri dataset e che regolarizza a livello legale la condivisione e la modifica di materiale. Tutte le immagini sono state collezionate da Flickr (altro tratto distintivo, nonostante altri dataset reperiscano le immagini dalla stessa fonte): dispone di 30.1M di etichette per 19.8mila concetti, 15.4 box di riconoscimento per 600 classi di oggetti e 375 mila relazioni che riguardano 57 classi. Le immagini nello specifico raffigurano scene complesse con mediamente 8 oggetti per foto. La prerogativa di OpenImages è stata quella di fornire immagini in diverse scale di qualità e molto varie tra loro e in particolare il dataset differisce per 3 elementi chiave rispetto agli altri concorrenti, due dei quali già elencati: la CC-BY License, il collezionamento da Flickr che lo rende ampio mentre il terzo tratto distintivo riguarda le immagini che non sono ottenute mediante la ricerca con tag specifici ma sono riferite a della classi naturali non definite, essendo scene ampie con più oggetti facilmente distinguibili o meno tra loro.
Dopo aver collezionato le immagini queste sono copiate in due risoluzioni:

- 1600HQ con massimo 1600pixel sul lato lungo e 1200 sul lato corto.
- 300K con immagini con massimo 300.000 pixel.

Subito dopo vengono estratti dalle foto i principali metadati e solo successivamente rimosse le foto con contenuti inappropriati, rimossi i duplicati e le foto maggiormente condivise su Internet. Alla fine di questi processi di pruning, viene creata una partizione in Train Images, Validation Images, Test Images.

4.2.9 Kitti Dataset

http://www.cvlibs.net/datasets/kitti/index.php

Il **Dataset Kitti** [121] [122] [123] [124] è frutto di un progetto dell'Università di Karlsruhe, in collaborazione con l'Istituto Tecnologico Toyota di Chicago, che sfruttando la piattaforma utilizzata per il progetto del gruppo di ricerca AnnieWAY (che sta sviluppando una auto dalla guida autonoma) ha equipaggiato una auto con due camere: una ad alta risoluzione e una in scala di grigi al fine di catturare immagini da aggiungere al dataset durante degli spostamenti nella zona della città di Karlsruhe.
Nell'ambito riconoscimento oggetti risultano essere disponibili 7481 immagini di training e 7518 immagini di test per un totale di 80.256 oggetti etichettati. Le immagini sono salvate in PNG e i Dataset forniti sono disponibili per soli ed esclusivi studi accademici.

4.2.10 CIFAR10 AND CIFAR 100

http://www.cs.toronto.edu/~kriz/cifar.html

CIFAR-10 e CIFAR-100 [125] sono un sottoinsieme di immagini raccolte da Alex Krizhevsky, Vinod Nair, e Geoffrey Hinton. CIFAR-10 contiene 60.000 immagini 32x32 divise in 10 classi equamente divise con 50.000 immagini di train e 10.000 immagini di test.
CIFAR-100 invece ha 100 classi da 600 immagini, con 500 immagini di train e 100 di test. Le 100 classi sono divise in 20 superclassi.
Oggi è uno dei dataset maggiormente usati per il testing delle reti neurali.

4.2.11 iNaturalist Species Detection Dataset

https://github.com/visipedia/inat_comp

iNaturalist Dataset [126] consiste in un set di 859 mila immagini di oltre 5mila specie di piante ed animali. Le immagini spesso ritraggono anche lo stesso tipo di specie, però in contesti diversi, in situazioni diverse, in posti diversi del mondo e sono collezionate con differenti camere permettendo di averle con diverse qualità.
Il focus di questo dataset è incentrato sulle immagini di training: se riducessimo il numero di immagini di training per categoria, per la classificazione perderemmo in termini di prestazioni, per cui si è cercato di fornire più immagini riguardanti la stessa specie in modo da essere performanti nelle prestazioni di classificazione e comunque in modo da adattarsi al principio naturale della biodiversità visto che realmente esistono milioni di specie tra piante ed animali. Questo fa sì che serva uno strumento di predizione che sia accurato e che copra in larga scala la biodiversità globale.
La base del dataset è una raccolta di immagini ed etichette, reperite dal sito scientifico

iNaturalist che permette di mappare e condividere osservazioni fotografiche delle biodiversità nel globo terrestre: ogni osservazione deve essere corredata però di data, luogo, immagini, ed etichette relative alle/alla specie presente in foto.
Nel dataset vi sono quasi 579.184 immagini di training e 95.986 immagini di validazione: per il training set la distribuzione delle immagini segue la frequenza di osservazione delle categorie fornita da iNaturalist quindi non vi è una distribuzione uniforme delle immagini per ogni categoria.

4.2.12 Places DataSet

http://places.csail.mit.edu/
https://github.com/CSAILVision/places365

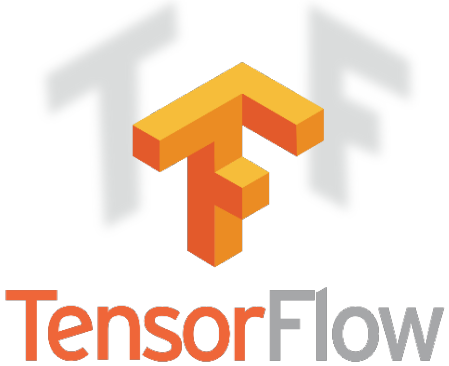
Il laboratorio del Mit di Computer Science e Intelligenza artificiale produsse questo dataset basato sulle scene e chiamato proprio **Places** [127] che consta di 205 categorie di scene e 2.5 milioni di immagini categorizzate.

4.2.13 Altri dataset

- **Oxford Flowers102Dataset**:
  http://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html
- **STL-10**:
  https://cs.stanford.edu/~acoates/stl10/
- **NORB**:
  https://cs.nyu.edu/~ylclab/data/norb-v1.0/
- **National Data Science Bowl Competition**:
  http://www.datasciencebowl.com/
- **COIL 20/100**:
  http://www.cs.columbia.edu/CAVE/software/softlib/coil20.php
  http://www.cs.columbia.edu/CAVE/software/softlib/coil-100.php
- **MIT-67 scene dataset**:
  http://web.mit.edu/torralba/www/indoor.html
- **Caltech-UCSD Birds-200 dataset**:
  http://www.vision.caltech.edu/visipedia/CUB-200-2011.html

4.3 Framework

4.3.1 Tensorflow



https://www.tensorflow.org/

TensorFlow [128] è uno dei più famosi framework open source per svolgere attività di machine learning e anche grazie alla sua ampia flessibilità ed adattabilità è stato adottato da molte aziende tra cui Airbnb, CocaCola, DeepMind, Airbus, Amd, Intel, Nvidia, Twitter, IBM, Google.
Google nello specifico ne ha fatto uso per l'ampia capacità di comprendere e processare linguaggi naturali, classificazione di testi, immagini, riconoscimento di scrittura manuale ed è l'azienda che porta avanti il suo sviluppo.

Funzionalità di TF

TensorFlow viene catalogato come uno dei tools più semplici e di facile comprensione nel momento in cui ci si approccia per la prima volta al Deep Learning, grazie all'ampia documentazione fornita ma anche perché usa di base Python come linguaggio e perché, come detto in precedenza, è supportato da Google e dispone di una ampia comunità di ricercatori che sviluppano continuamente Api aggiornate e migliorate in modo da mantenere sempre aggiornata la piattaforma allo stato dell'arte attuale nell'ambito del machine learning.
Tensor Flow si compone di 2 tools fondamentali: il **TensorBoard** che permette la visualizzazione dei dati nel modello di rete e le relative performance e il **TensorFlow Serving** utilizzato per lo sviluppo di nuovi algoritmi.
In particolare, TensorBoard è un grosso punto di forza rispetto ad altri framework, visto che gli altri sono costretti a sfruttare la libreria **Matplotlib**; ultimamente si è cercato di adattare TensorBoard anche per PyTorch ma non è un progetto nativo del tool.
La nuova implementazione di TensorFlow Lite ne apre ancor di più gli orizzonti di utilizzo.
Le Api di alto livello di cui si fa uso sono le seguenti:

- **Keras**, usata per costruire ed allenare i modelli di Deep Learning. È una Api con interfaccia semplice, ottimizzata per i casi di utilizzo comune e che fornisce anche dei feedback nel caso di errori dell'utente.
  È caratterizzata da dei blocchi connessi configurabili, collegabili tra loro ma anche estendibili nel caso si volesse proporre una nuova idea di modello.

- **Eager Execution**, è un ambiente/piattaforma di programmazione per ricerca e sperimentazione ma anche per valutare le operazioni in modo immediato senza il passo successivo della creazione di un grafo.
  L'uso di Python è la base per questa Api, in quanto permette di iterare dati e modelli scritti nel suddetto linguaggio fornendo la possibilità di debuggare in maniera rapida modelli in esecuzione e controllare il flusso stesso di questi, evitando l'uso del controllo flusso dei grafi.
  Questa Api supporta tantissime operazioni di Tensor Flow e l'accelerazione GPU ma l'unica nota negativa è che l'uso di diversi modelli incrementa l'overhead se l'eager execution è abilitata.

- **Importing Data** che corrisponde alla pipeline di input per i dati in ingresso nel nostro programma TensorFlow.

- **Estimators** che è una Api che fornisce un pacchetto completo di modelli per training e produzione dati.

TensorFlow dispone anche di una serie di Api di basso livello dette **TensorFlow Core**:

- **Introduction** che indica come muoversi tra tutte le Api di basso livello, mostrando come gestire il programma di base o come gestire le operazioni di sessione a runtime, oppure come usare le Api di alto livello o come costruire in nostro ciclo di training o semplicemente come usare uno di quelli forniti da Estimators.

- **Tensors** che espone come creare, accedere e manipolare i tensori cioè gli elementi fondamentali di TensorFlow: infatti tutto il sistema è basato sulla definizione ed esecuzione di operazioni che coinvolgono proprio i tensori, siano essi dei vettori o delle matrici generiche. Quando si scrive un programma per TensorFlow l'operazione basilare è quella di manipolare i tensori infatti il programma lavora formando un grafo di oggetti TensorFlow. Questi possono essere di tipo float32, int32, string o uno shape, che corrisponde al numero delle dimensioni che ha; fondamentale è che ogni tensore abbia elementi tutti dello stesso tipo. Vi sono altri tipi speciali di tensori quali variabili, costanti, placeholder o Sparse Tensor.

- **Variables**: una variabile TensorFlow è il miglior modo per rappresentare lo stato condiviso o persistente del nostro programma.

- **Graphs and Sessions**: viene usato un grafo per per rappresentare lo stato computazionale delle operazioni in termini di dipendenza proprio tra le singole operazioni. DataFlow è un comune modello per monitorare la computazione delle operazioni nelle unità: ogni nodo corrisponde ad una unità operativa. Viene così definito un grafo che corrisponde esattamente ad una sessione operativa.

- **Save and Restore**: espone come salvare e ripristinare variabili e modelli.

- **Ragged Tensor**: permette di capire come usare questi elementi che non sono altro l'equivalente di liste senza dimensione definita annidate. Questi sono usati in molte operazioni di TensorFlow tra cui quelle matematiche.

TensorFlow fornisce anche un Debugger mentre la versione Lite fornisce delle tecniche di ottimizzazione per le versioni mobile ed embedded.
Altro aspetto da considerare di TensorFlow è la performance, fondamentale quando si allena un modello di machine learning: a tale scopo vengono forniti documenti e descrizioni

per migliorare l'input dei dati, o per migliorare la configurazione o semplicemente per implementare le tecniche migliori di ottimizzazione del codice scritto. **"Distribution Strategy"** è un'altra Api di TF ed è un modo semplice per distribuire la fase di training tra più device o macchine. Permette agli utenti di usare modelli esistenti e allenare codice con poche modifiche, nel caso si volesse proprio fare allenamento in modo distribuito.

Aspetti Pratici di Tensorflow

TensorFlow (TF) è uno dei 3 framework per cui sarà fornito supporto in questa trattazione e per cui sono stati scritti diversi script implementativi atti a completare i task evidenziati nell'introduzione.
Per interfacciarsi fin da subito con TensorFlow è stata sfruttata l'ampia guida fornita sia sul sito ufficiale sia nel repository ufficiale di GitHub. Oltretutto quando ci si avvicina ad un framework di machine learning è buona norma valutare il proprio sistema e verificare se tra i prerequisiti hardware del sistema vi sia una GPU, che permette di avere prestazioni migliori in qualsiasi attività di machine learning visto e considerato che si ha a che fare con operazioni dal costo computazionale alto.
GPU (graphics processing unit) e **TPU (tensor processing unit)** possono ridurre notevolmente le tempistiche di esecuzione di un singolo step di training oltre che l'utilizzo anche di una pipeline efficiente che invia i dati alla fine di ogni step in maniera tempestiva.
L'Api "tf.data" aiuta a rendere flessibile ed efficiente la pipeline che in input su TF prevede le seguenti fasi:

- Estrazione.
- lettura dallo storage.
- trasformazione degli input mediante l'uso della CPU.
- caricamento dei dati su un device GPU o TPU.

Questa Api fornisce un modello a blocchi per standardizzare il processo.

TensorFlow Datasets

TensorFlow Datasets fornisce una serie di dataset pubblici attraverso l'istruzione "tf.data.Datasets" e in particolare, attraverso il seguente import:

```
import tensorflow\_datasets as tfds
```

Grazie a questa funzionalità può essere costruito un Dataset "custom" in maniera guidata. Per ulteriori approfondimenti nella costruzione dettagliata di un dataset si può fare riferimento alla documentazione fornita nel seguente repository:
https://github.com/tensorflow/datasets.

TF offre supporto per una vasta quantità di svariate tipologie di dataset, tra cui alcuni di quelli dettagliati nella precedente sezione sui dataset.

Le Api della Community TF e il Model-ZOO

TensorFlow è dotato di una Community che si poggia su un repository di GitHub suddiviso in due grosse macro sezioni:

- Una sezione ufficiale, nella quale sono degli sviluppatori accreditati a postare Modelli Architetturali già allenati ed Api di esempio di alto livello: tutto il materiale di questa sezione è sottoposto a testing continuo, aggiornamenti periodici.
  Fondamentalmente può essere definita come un repository ufficiale, clonabile nella nostra cartella di lavoro, e per il quale si possono installare le relative dipendenze.

- Una sezione relativa a ciò che viene proposto dai ricercatori, che quindi non ha nulla a che vedere con il team ufficiale di sviluppo del framework.
  In particolare, sono i ricercatori stessi a richiedere di poter sottomettere del materiale, e sono loro stessi a mantenere testato ed aggiornato ciò che postano. La directory è reperibile sotto il nome Research e contiene modelli di rete già allenati, Api di utilizzo diretto del framework, esempi di utilizzo per allenamento, transfer learning o esempi di valutazione delle prestazioni sotto forma di tutorial o notebook Jupyter.
  Tra questi di nostro interesse sarà la directory "Object Detection" e la directory "Slim" che ci aiuteranno a performare i nostri task.

Repository Ufficiale

Facendo una veloce carrellata dei vari path ci si accorge facilmente che non tutti i modelli architetturali sono utili alla nostra trattazione e quindi sarà preso in considerazione solo ciò che concerne le CNN per immagini e in particolare:

- **KERAS_APPLICATION_MODELS**

  Questa risulta essere una cartella ponte verso i modelli keras per la classificazione delle immagini; le principali reti sono:
  Xception, VGG16, VGG19, ResNet50, InceptionV3, InceptionResNetV2, MobileNet, DenseNet, NASNet. Vengono inoltre forniti degli script diretti attraverso cui fare un'analisi computazionale usando nello specifico dei dataset ridotti:

  ```
  python benchmark_main.py --model resnet50 -eager
  ```

  Con questo comando ad esempio verrà lanciata l'esecuzione eager della rete "Resnet50" e quindi sostituendo dopo –model il nome del modello da testare verrà provata qualunque altra tipologia di architettura disponibile.
  Volendo usare la "distribution strategy" (non supportata da eager) il comando sarebbe:

  ```
  python benchmark_main.py --model resnet50 -dist_strat
  ```

  L'eventuale aggiunta del parametro –callback fornisce la possibilità di usare una lista di callback aggiuntive.

- **ResNet**:

  https://github.com/tensorflow/models/blob/master/official/resnet

  Contiene una implementazione della Rete ResNet allenata su ImageNet per TF. I dettagli relativi alle ResNet sono stati visti già nella sezione esplicativa delle reti mentre ciò che risulta interessante è l'indicazione di alcuni modelli già pre-allenati

di Resnet-50. I modelli fruibili fanno riferimento a dei checkpoint di allenamento, prodotti dall'Api Estimator di TF oppure modelli in versione portable che occupano meno spazio di storage. Il fatto che siano proposti dei checkpoint rende non sicuro l'utilizzo con le versioni successive di codice del framework considerato che possono variare dei dettagli implementativi delle reti come ad esempio che le ResNet v1 e v2 sono allenate in precisione floating point 16 e 32 o che i modelli salvati accettano sia tensori, che immagini in formato jpg in input e sono distinguibili in due tipologie: **NCHW** con convoluzione sul primo canale e ottimizzate per GPU e **NHWC** ottimizzati su CPU.

TF Slim

https://github.com/tensorflow/tensorflow/tree/master/research/slim

Slim [129] è una libreria leggera per definire, allenare, e valutare modelli in TensorFlow. Sarà ora analizzata dettagliatamente e testata nei capitoli successivi. I suoi componenti possono essere facilmente mixabili con quelli tradizionali di TF ma a livello implementativo è composta da diversi elementi indipendenti da TF quindi in qualche modo è definibile come una vera e propria Api. Si compone di diversi parti funzionali:

- **TF Slim Data** è una libreria per facilitare la lettura da vari formati standard di utilizzo comune ed è composta a sua volta da diversi layer in modo da permettere flessibilità in caso di supporto in lettura per file multipli. Ha due componenti principali, uno che specifica come il dataset è rappresentato e uno che fornisce le istruzioni per ottenere i dati.
  A livello puramente implementativo TF Dataset è una tupla che incapsula:
  data_sources cioè la lista di path che compone il dataset, **reader** che corrisponde al TFReader appropriato per il tipo di data_sources, **decoder** che indica la classe usata per decodificare il contenuto del dataset, **num_samples** che è chiaramente il numero di samples del dataset, **items_to_description** cioè la mappa degli item forniti. Operativamente, il dataset è letto aprendo i file specificati nel data_sources, mediante la classe reader che sfrutta il decoder indicato per richiedere una lista di item che saranno ritornati come Tensori.

- **arg_scope** fornisce un nuovo "scope" che permette all'utente di definire gli argomenti di default per delle specifiche operazioni.

- **evaluation**: contiene le routine per valutare i modelli.

- **Layers**: contiene i livelli di alto livello per costruire i modelli.

- **Learning**: contiene le routine per allenare i modelli.

- **Losses**: contiene le funzioni di loss più usate.

- **Metrics**: contiene i parametri di valutazione metrica più diffusi.

- **Nets**: contiene le definizioni delle reti più popolari.

- **Queue**: fornisce un gestore di contesto per per iniziare e chiudere agevolmente e in modo sicuro la QueueRunners.

- **Regularizers**: contiene i regolarizzatori del peso.

- **Variables**: fornisce dei wrapper per la creazione e manipolazione di variabili.

I modelli possono essere definiti usando TF SLIM mediante la combinazione di Variables, Layers e Scope. Le variabili (ad esempio la variabile weights) in TF, quando vengono generate, richiedono dei valori predefiniti o comunque un meccanismo di inizializzazione quindi è necessario conoscere i requisiti implementativi del modello da generare o sfruttare. TFSlim per questo fornisce un set di funzioni che possono permettere di definire delle variabili (nel file variables.py): queste dopo essere state create verranno inserite nel modello come parte integrante.
Un layer che sia Convoluzionario o Fully Connected, è qualcosa di astratto, che compie diverse operazioni e che può avere delle variabili associate: ad esempio un layer convoluzionario crea dei pesi e delle variabili "bias", fa una convoluzione dei pesi con il risultato del livello precedente, poi aggiunge le bias e applica una funzione di attivazione. Quanto detto normalmente viene scritto con una serie di operazioni mentre con TF-Slim può essere sostituito con l'invocazione di una singola riga di codice, grazie all'utilizzo degli "scope". Gli scopi, altra grande novità, reperibile anche nella versione TF ma solo relativamente a nomi e a variabili, sono una interessante funzionalità fornità in più per l'utente: permettono di semplificare il codice ed evitano anche di indicare per ogni layer i valori passati.
I modelli di rete in TF vanno allenati e questo richiede anche il codice e la conoscenza l'architettura quindi del modello stesso, la scelta di una funzione di perdita e di un gradiente e la definizione di una routine di training che iterativamente svolge le operazioni necessarie e definite. TF-Slim invece provvede anche a definire delle funzioni che eseguono il training e la valutazione dei modelli. Allenato il modello questo potrà essere ripristinato facendo tornare le variabili ad uno stato precedente valido, attraverso dei meccanismi come tf.train_Saver() che permettono di ripristinare una o tutte le variabili.
A questo punto, nel naturale processo di una rete neurale si giunge a ciò che maggiormente ci interessa: valutare un modello. Questo processo è caratterizzato dalla scelta di un set di metriche di valutazione utili a classificare le performance rispetto ad un altro modello. Al fine di avere delle valutazioni obiettive l'operazione va svolta ripetutamente. La metrica relativamente alla misurazione delle prestazioni in TF SLIM è definita attraverso un set di operazioni (dentro metric_ops.py) che permettono di svolgere la valutazione in maniera intuitiva: vengono inizializzate le variabili usate, vengono eseguite le operazioni obbligatorie per calcolare la metrica e poi se vi sono altre operazioni finali facoltative vengono eseguite; a questo punto TF SLIM offre il modulo di valutazione (evaluation.py) che unito a metrics_ops.py saranno punto di partenza per scrivere modelli di valutazione validi [130]. Per gli utenti sono già stati predisposti all'interno del repository degli script mirati a perfezionare CNN già allenate o script per scaricare i dataset standard, script per convertirli in dataset per TensorFlow (in formato TFRecord) e soprattutto codice di esempio per capire come lavora TF Slim nell'ambito della classificazione delle immagini.
Quindi viste le premesse fatte, questa sarà una delle Api di rilevante interesse per la nostra ricerca. In parole povere, grazie a Tensorflow Slim e a ciò che offre, si può, mediante la chiamata di alcune tra le funzioni descritte, effettuare una operazione utile al nostro interesse, allenare la rete. costruire un dataset: per tale motivo sarà nostro compito costruire degli script che richiamino tali funzioni in modo da sfruttarle a nostro piacimento.

Il repository di TFSlim ci fornisce diversi anche modelli scaricabili ed utilizzabili per i possibili script di **Inference** che saranno sviluppati. Per ogni modello sono disponibili varie versioni, più o meno datate e quindi più o meno precise: tra quelle di spicco queste sono le versioni più performanti.

- **Inception V3**
- **Inception V4**
- **Inception Resnet V2**
- **Vgg 16**
- **Vgg 19**
- **Mobile Net v1 224 (v1.0)**
- **Mobile Net v2 224 (v1.4)**
- **NasNet-A Mobile 224**
- **PnasNet-5 Mobile 224**

Api: TF Object Detection

https://github.com/tensorflow/models/tree/master/research/object_detection

TensorFlow Object Detection è una Api open source creata per realizzare, allenare e testare modelli di riconoscimento e localizzazione oggetti e quindi sarà una delle applicazioni centrali della nostra tesi visto che sarà nostro obiettivo sfruttare dei modelli capaci di completare il suddetto task.
Questa Api si pone sopra Tensorflow sfruttando le sue librerie e i suoi moduli oltre che Protobufs per configurare i modelli ed allenare i parametri. Queste di seguito sono le reti pre-allenate fornite:

- **SSD_MobilenetV1** allenata sul dataset COCO in diverse varianti
- **SSD_Mobilenetv2** allenata sul dataset COCO on diverse varianti.
- **SSD_Mobilenetv2** allenata su Open_images.
- **SSD ResNet50**.
- Diverse varianti di **Faster_R-CNN** allenate sul dataset Coco, Kitti Dataset, Open_images, i-Naturalist.
- Diverse varianti di **Mask_R-CNN** allenate sul dataset Coco.

Alla seguente pagina [131] sono disponibili i link relativi per reperire i modelli di nostro interesse. TF Object Detection così come TF Slim dispone di alcune librerie, funzionali al nostro scopo, che pertanto saranno sfruttate e chiamate all'occorrenza nei nostri script per permettere di completare il task della localizzazione unito alla classificazione degli oggetti.

4.3.2 CAFFE

http://caffe.berkeleyvision.org/

Caffe [132]: è un framework che supporta interfacce C, C++, Python, Matlab e che lavora anche da linea di comando. È stato sviluppato dal Berkeley Vision and Learning Center (BVLC), con il contributo di un'intera community.
Caffe potenzia progetti di ricerca accademica, prototipi di startup e applicazioni industriali visive, vocali e multimediali su larga scala. È utilizzato moltissimo con modelli di Reti Convolutive e risulta essere molto veloce.
Uno degli elementi di particolare rilevanza, derivato dall'uso della libreria C++, è la possibilità di utilizzare liberamente le reti disponibili nel repository "Caffe Model Zoo" che sono pre-allenate e quindi pronte all'uso, spesso anche con funzioni di chiamata già predisposte dal framework.
Nell'ambito di nostro interesse cioè quello del processamento di immagini è uno dei framework più usati, soprattutto perché può grazie alla sua velocità di processamento, analizzare oltre 60 milioni di immagini in un giorno con una singola scheda Nvidia K40 GPU.
Caffe di contro non supporta la fine granularità dei livelli come TensorFlow e non vi e' molto supporto per il modellamento delle reti: i layer sono strutturati e definiti principalmente ai livelli bassi di programmazione e nel caso si vogliano fare delle modifiche per ottenere magari livelli con granularità maggiore si va incontro a particolari difficoltà.
Come visto già con TensorFlow, per sfruttare l'accelerazione GPU è necessario installare il supporto CUDA di NVIDIA e le relative dipendenze e quindi chiaramente servirà l'installazione della versione di Caffe con supporto CPU.

Caffe Repository

Caffe è anche una community infatti sul repository GitHub di riferimento di seguito linkato è possibile reperire una grande quantità di materiale utile.
https://github.com/BVLC/caffe.
Sono necessarie svariate dipendenze, appositamente indicate nella guida all'installazione presente nel repository appena citato: tra i più significativi c'è BLAS (http://www.netlib.org/blas/) che offre delle routine per compiere operazioni matematiche con vettori e matrici.
Sono forniti come anticipato dei modelli pre-allenati che vengono scaricati direttamente nella directory models/¡nomemodello¿, lanciando lo script dalla cartella principale della repo il seguente comando:
scripts/download_model_binary.py dirname dove "dirname" è specificatamente uno dei modelli proposti:

- **Modello AlexNet** allenato su ILSVRC2012 con variazioni leggere che in letteratura è definito come REFERENCE_CAFFENET: dirname diventa models/bvlc_reference_caffenet.

- **Modello AlexNet** allenato ILSVRC 2012 in maniera tradizionale detto ALEXNET dirname diventa models/bvlc_alexnet.
- **Modello R-CNN ILSVRC-2013**:
  dirname diventa models/bvlc_reference_rcnn_ilsvrc13.
- **Modello GoogLeNet**: dirname models/bvlc_googlenet.

Caffe Model Zoo

Alla seguente pagina invece [133] vi sono una svariata quantità di modelli, selezionati qui di seguito, scegliendo come criterio principale il fatto che lavorassero sui task di nostro interesse e selezionando tra le svariate versioni quelle più prestanti o quelle meno pesanti in termini di storage.

- **Network in Network model**.
  La community fornisce due modelli: il modello pre-allenato su Imagenet su 4 livelli che risulta essere di dimensioni ridotte grazie alle modifiche sui fully connected layer, che ne ha ridotto oltre che le dimensioni anche il tempo di allenamento; il secondo modello fornito è il modello pre-allenato su Cifar10 su 3 livelli.
- Modelli proposti nel seguente documento **"Return of the Devil in the Details: Delving Deep into Convolutional Nets"** [134].
  Tra le varie reti proposte si preferirà quella allenata sul dataset ILSVRC-2012 su immagini 224x224 che è una rete VGG, denominata VGG_CNN_F che secondo il documento è quella più accurata.
- **Rete VGG usato nel ILSVRC-2014**.
  Sono proposti i modelli con 16 layer e 19layer, i modelli migliori proposti per la competizione di quell'anno.
- **Modelli proposti dal MIT su DATABASE Places**.
  Pagina relativa al progetto: http://places.csail.mit.edu/
  Per allenare le seguenti reti è usato principalmente il dataset Places a cui viene accoppiato anche l'ILSVRC-2012 di ImageNet. Viene fornito un modello denominato Hybrid, una AlexNet e una GoogLeNet.
- **Modello proposto da Princeton di una GoogLeNet allenata su GPU**.
  Pagina relativa al progetto: http://3dvision.princeton.edu/pvt/GoogLeNet/.
  I modelli disponibili sulla pagina del progetto sono allenati su ImageNet e su Places-Database. Gli elementi del file prototxt devono avere, per i campi cls2_fc2 e cls3_fc, il valore num_output = 1000, in modo da correttamente nelle applicazioni.
- **Modello ottimizzato per Oxford Flowers Database**.
  Il modello di rete prende spunto da quello di riferimento di Caffe cioè la AlexNet allenata su ILSVRC 2012, ma in questo caso è ottimizzato per l'Oxford Flowers Database che ha per l'esattezza 102 output, come il numero di categorie di fiori del set.
- **Parsenet** [100].
  Modello allenato su Dataset Pascal.
- **Modello Faster_R-CNN**.
  Questo modello è allenato su PASCAL VOC DATASET 2007.

- **Modelli ResNet**.
  GitHub Repository: https://github.com/KaimingHe/deep-residual-networks
  I modelli allenati sono stati proposti per il contest ILSRVC e per quello Coco. Questo modello vinse in definitiva le seguenti competizioni: ImageNet classification, ImageNet detection, ImageNet localization, COCO detection, e COCO segmentation. Il progetto oggi è portato avanti da MRA (Microsoft Research Area).

- **Modello SqueezeNet**.
  SqueezeNet viene proposta in diverse versioni tutte allenate con ImageNet:
  a) SqueezeNet 1.0.
  b) SqueezeNet 1.1.
  c) SqueezeNet 1.0 Deep-Compression: dimensioni ridottissime, circa 660KB molto più piccola della AlexNet ma con la stessa accuratezza teorica.
  d) SqueezeNet Residual: viene ottenuta aggiunge un bypass layer alla SqueezeNet 1.0.

- **Modello Inception v2 con Batch Normalization** [20].
  Nello specifico questo selezionato, è stato allenato su ImageNet.

Questi evidenziati appaiono come i modelli più salienti che la community di Caffe mette a disposizione ma in realtà sono presenti altri modelli, molti dei quali a noi non utili o che non dispongono di tutti i file necessari per un utilizzo immediato poichè difettano del file deploy.prototxt, che deve essere fornito da chi allena la rete, dal momento che contiene dei parametri settati in fase di training.
Altre reti pre-allenate sono reperibili alla pagina:

http://www.vlfeat.org/matconvnet/pretrained/

4.3.3 Torch e PyTorch

http://torch.ch/

http://pytorch.org/

Torch è un framework che offre ampio supporto per il machine learning. Il suo utilizzo è diffuso tra diversi colossi mondiali quali Facebook, Twitter o Google. È prevalentemente usato con librerie C/C++ ed implementa le librerie CUDA.
PyTorch invece è uno dei framework più in voga al momento, è considerato quasi il competitore di TensorFlow ed è essenzialmente una porta verso Torch; è usato per costruire reti neurali e eseguire la computazione specialmente nei casi di alta complessità.
Sviluppato dal Fair (Facebook Al Research) usa Python a differenza di Torch ed è caratterizzato da un processo di modellamento molto più semplice rispetto a quello eseguito da Torch. Come TensorFlow l'elemento fondamentale di PyTorch sono i tensori, matrici multidimensionali di numeri, necessarie per il calcolo matematico in python; PyTorch permette di produrre modelli molto più intuitivi e facilmente studiabili, nonostante alle sue spalle vi sia una comunità di sviluppo minore ma comunque in crescita esponenziale. Nel capitolo successivo saranno visti diversi esempi che girano su PyTorch.

L'introduzione appena fatta ha sottolineato che si tratta di uno dei framework sui cui in termini di ricerca si sta puntando molto.
In questa pagina del sito ufficiale: [136] viene data la possibilità di scegliere le proprie impostazioni di installazione, se avvalersi del supporto Cuda o semplicemente il tipo di sistema operativo. Come per gli altri framework esiste un apposito repository nel quale reperire modelli di rete allenati: [137].

Pytorch: componenti e librerie

Pytorch a livello strutturale si compone e avvale di diversi elementi:

- Una libreria contenente strutture dati per tensori multidimensionali e che al suo interno dispone delle principali operazioni matematiche sui tensori e la serializzazione di questi per i più svariati utilizzi: questo package è chiamato **Torch**.
  https://pytorch.org/docs/stable/torch.html
- Uno stack di compilazione per creare modelli serializzabili ed ottimizzabili a partire dal codice di PyTorch detto **torch.jit**.
  https://pytorch.org/docs/stable/jit.html
- Una libreria che fornisce classi e funzioni di automatica differenziazione dei valori scalari e che supporta ogni operazione sui Tensori: **torch.autograd**.
  https://pytorch.org/docs/stable/autograd.html
- Una libreria sulle reti neurali: **torch.nn**.
  https://pytorch.org/docs/stable/nn.html

- Una libreria python che permette il multiprocesso e quindi la condivisione della memoria dei tensori tra processi: **torch.multiprocessing**.
  https://pytorch.org/docs/stable/multiprocessing.html
- Una libreria di utilities che contiene funzioni e processi come il DataLoader molto utili per l'utilizzo del framework: **torch.utils.data**.
  https://pytorch.org/docs/stable/data.html

I modelli architetturali in PyTorch

Per testare il framework è necessario come sempre un modello allenato: di questo dopo l'allenamento viene salvato un "checkpoint" che sarà utilizzato poi per l'Inference. I modelli sono definiti da una classe che estende una classe "Module" mentre tutti i componenti sono presenti nel package torch.nn che sarà necessario importarla nelle dichiarazioni quando si vorrà scrivere uno script. Il caricamento e l'utilizzo del set di dati invece è molto semplice e necessita del pacchetto "torchvision". Dato che la costruzione del dataset e l'allenamento non svolgono un ruolo principale per la nostra tesi, si passerà subito ad analizzare nel dettaglio l'interazione con il modello allenato, e l'utilizzo di questo per ottenere delle predizioni su nuove immagini; i passaggi da eseguire sono i seguenti:

- Definire e istanziare lo stesso modello che è stato allenato.
- Caricare il checkpoint nel modello.
- Caricare un'immagine di test dal nostro filesystem, che è quella che verrà valutata.
- Lanciare questa immagine sul modello.
- Ottenere e leggere la migliore predizione.
- Convertire la predizione numerica in una classe categorica.

Questi più o meno sono dei passaggi standard visti anche negli altri framework, con la sola differenza che in qualche modo torch fornisce una sorta di aiuto distribuendo, per i modelli maggiormente usati, delle librerie con le relative funzioni istanziabili. Sarà comunque necessario il seguente import:

```
import torchvision.models as models.
```

Alcuni tra i modelli disponibili sono i seguenti:

- **resnet18 = models.resnet18()**.
- **alexnet = models.alexnet()**.
- **vgg16 = models.vgg16()**.
- **squeezenet = models.squeezenet1_0()**.
- **densenet = models.densenet161()**.
- **inception = models.inception_v3()**.
- **googlenet = models.googlenet()**.

Sono forniti anche in versione pre-allenata e sono importabili e richiamabili nel modo seguente; la prima volta il modello sarà automaticamente scaricato.

- **resnet18 = models.resnet18(pretrained=True)**.
- **alexnet = models.alexnet(pretrained=True)**.
- **squeezenet = models.squeezenet1_0(pretrained=True)**.
- **vgg16 = models.vgg16(pretrained=True)**.
- **densenet = models.densenet161(pretrained=True)**.
- **inception = models.inception_v3(pretrained=True)**;
- **googlenet = models.googlenet(pretrained=True)**.

Altri Pre-Trained Model per PyTorch

Muovendoci tra i vari repository che trattano PyTorch sul Web, è apparsa di importante rilevanza tecnica la seguente:
https://github.com/Cadene/pretrained-models.pytorch
Vengono proposti diversi modelli pre-allenati e un supporto completo per il relativo utilizzo in termini di funzioni richiamabili: dopo l'installazione, dettagliata sul repository, dagli script o programmi sviluppati si potrà semplicemente fare il relativo import di **import pretrainedmodels** per usufruire di un vasto numero di reti allenate tra le più prestati del momento e del relativo codice in python per l'utilizzo.
Attraverso il seguente semplice script python si può verificare quali sono i parametri da usare per l'eventuale utilizzo indicando tra gli apici uno dei nomi elencati come di seguito:

```
import pretrainedmodels
print(pretrainedmodels.pretrained_settings['densenet201'])
```

4.3.4 Caffe2

https://github.com/caffe2

Caffe2 è un framework che discende dal ben più famoso Caffe con la semplice differenza che questa è una versione leggera, flessibile, modulare e veloce. All'interno del repository, sopra linkato, si possono trovare molti modelli pre-allenati ma soprattutto degli script che permettono la conversione di un modello da Caffe a Caffe2. Dispone infatti di una discreta "ModelZoo" alla pagina [138] mentre nella seguente [139] viene fornita una ampia quantità di tutorial di utilizzo per gli utenti. In particolare, però al momento quello che risulta essere di rilevante interesse è che sono utilizzabili le librerie di Caffe2 mediante PyTorch e quindi mediante l'installazione del solo Pytorch.

4.3.5 Keras

https://keras.io/

Keras è' una libreria di Reti Neurali molto minimalista, supportata da una interfaccia Python mentre il framework è sfruttabile sia per le RRN e le CNN.
La libreria come già visto è usata anche su TensorFlow, di cui è anche una Api di alto livello mirata alla costruzione di reti neurali.
Non verrà approfondita direttamente nessuna applicazione del framework ma solo un esempio di script di testing di rete neurale già allenata, disponibile nel seguente repository: [140].

4.3.6 CNTK: The Microsoft Cognitive Toolkit

https://github.com/Microsoft/CNTK

Anche questo framework è open source e risulta essere molto simile a Caffe in quanto opera in modo efficiente su reti convolutive per il processamento di immagini, linguaggi parlati e testi ed è supportato anche da interfacce in C++, Python oltre che essere utilizzabile da linea di comando. Fornisce prestazioni di gran lunga migliori rispetto ad altri framework, se utilizzato su macchine multiple.
Gli user non devono programmare a basso livello per implementare nuovi layer complicati, cosa che era necessaria in Caffe. Supporta anche le RNN, ma soprattutto le reti di nostro interesse cioè le CNN, ed è utilizzabile anche su ARM. Per questo framework non verranno fatti ulteriori approfondimenti sull'uso pratico.

4.3.7 Altri Framework reperibili in Rete

Lasagne: https://lasagne.readthedocs.io/en/latest/
DL4J(DeepLearning4J): https://deeplearning4j.org/
Chainer: http://chainer.org/
DIGITS: https://developer.nvidia.com/digits
OpenDeep: http://www.opendeep.org
PuRine: https://github.com/purine/purine2
MINERVA: https://github.com/dmlc/minerva
MXNET: https://github.com/dmlc/mxnet

4.4 Detectron e Detectorch: Porting di Detectron per PyTorch

https://github.com/facebookresearch/detectron

Detectron è un software che implementa lo stato dell'arte degli algoritmi per riconoscere e localizzare gli oggetti e per fare segmentazione degli oggetti. È stato sviluppato in Python dal Facebook Al Research Team per lavorare su Caffe2. Questo team ha portato avanti svariati progetti in ambito ML come le Feature Pyramid Networks for Object Detection, le Mask R-CNN presenti anche in Detectron, e diversi altri progetti di Object Detection con svariate tecniche con l'obiettivo principale e dichiarato di fornire alta qualità e grandi performance. Detectron fornisce supporto per Mask R-CNN, RetinaNet, Faster R-CNN, RPN, R-FCN, e Fast R-CNN e permette l'utilizzo di altre architetture quali ResNext, ResNet, Feature Pyramid Networks, Vgg16 e diverse altre reti di recente progettazione.
Per fare il testing sulle reti richiesta molta capacità computazionele quindi l'uso della GPU è obbligatorio e in particolare, è necessario avere delle GPU molto performanti per non incorrere in errori, anche durante il testing su reti magari anche un po' più datate. Nel model zoo relativa [141] sono proposti diversi modelli pre-allenati e in particolare all'interno della cartella Configs [142] sono proposte diverse possibili configurazioni utili per fare partire **l'inference**, che come già detto è la pratica di testing delle reti, in modo diretto, fornendo delle immagini a nostra scelta in input, e settando dei parametri più o meno facoltativi e comunque di alto livello relativi all'allenamento.

4.4.1 Strumenti aggiuntivi per l'uso dei Framework

GPU E TPU

Introducendo TensorFlow è stato necessario parlare di GPU e TPU.
La GPU è un chip progettato da Google per lavorare anche con algoritmi di ML ed è un ottimo strumento per lavorare al meglio con TF o con un qualsiasi altro framework, laddove lavorare con un chip generico porterebbe maggiori difficoltà.
Si tratta di un **ASIC (application specific integrated circuit)** ed è più tollerante con precisione di calcolo ridotta, per cui richiede meno transistor per funzionare: questo fa sì che si possano richiedere più operazioni al secondo per ogni silicio, permettendo l'esecuzione di algoritmi più sofisticati e potenti.
L'idea di TPU, di contro, si avvicina a quella di un co-processore FPU (floating point unit). È risaputo anche che una singola esecuzione può essere distribuita su molteplici GPU, cosa che riesce a ridurre il tempo di training a durate di qualche ora rispetto alle tempistiche con CPU, che potevano raggiungere durate anche di molteplici giorni. L'elaborazione infatti negli ultimi anni è passata da un concetto di centralizzazione su singola CPU ad un concetto di co-processing CPU+GPU, così **CUDA**, che è l'architettura di elaborazione in parallelo proposta da NVIDIA è diventata parte integrante di moltissime GPU in commercio.

Docker

Docker in questo contesto è un altro strumento che è risultato utile per lo sfruttamento dell'esecuzione con supporto GPU: attraverso l'uso di "contenitori", si possono creare ambienti isolati per separare l'esecuzione di una operazione di Machine Learning dal resto delle operazioni in esecuzione sul sistema.
Quindi di conseguenza è chiaro che il nostro framework, qualunque esso sia, può essere lanciato dentro questo ambiente che permette di condividere le risorse principali del sistema. Docker è già configurato ad esempio per TensorFlow, framework sul quale è stato testato, ma va comunque scaricato e installato sul proprio sistema per poterlo utilizzare.

Jupyter

Jupyter è una applicazione di utilizzo comune per chi lavora nell'ambito della nostra ricerca o comunque sviluppa del codice in python: molti tutorial infatti sono fruibili in questo formato.
Questo permette di editare, lanciare e quindi eseguire degli script o dei pezzi di codice solo attraverso l'uso di un browser. I notebook, cioè i documenti in sé e per sé, sono sezionabili in parti di codice e in parti esplicative del codice quindi commenti sulle operazioni svolte. È anche possibile verificare l'esecuzione di un pezzo di codice a runtime o anche visualizzare i grafici ottenuti mendiante il plotting in python.

Capitolo 5

Scenari Applicativi

5.1 Inference e Scenari pratici

In questo capitolo, si abbandonerà il contesto prettamente tecnico-teorico per passare ad un contesto totalmente pratico.
Definiti i concetti di base, fatta una rassegna dei dataset e dei framework disponibili, delle Api utilizzabili e avendone conosciuto funzionamento, peculiarità, vantaggi e svantaggi, si è scelto su che direzione pratica proseguire, focalizzandosi su alcuni scenari applicativi da sviluppare. Sono stati prodotti per l'appunto degli script in Python che sfruttando le nozioni teorico tecniche apprese e le librerie conosciute permettessero di avere un riscontro visivo di quanto anticipato. Questo passaggio è definito **Inference**.
Le basi da cui partire sono: immagini e video e reti neurali pre-allenate reperibili sul web, o nei repository visti nel capitolo precedente; in output si avranno delle immagini contrassegnate con dei box e dei file esterni testuali con predizioni e categorie relative ai box nel caso di Localizzazione mentre nel caso della Classificazione avremo delle percentuali di predizione affiancate dalla relativa classe predetta tendenzialmente per i primi 5 risultati più alti. Alcune delle applicazioni sono state sviluppate in Python per TensorFlow, Caffe e PyTorch invece altre sono state testate ed approfondite, come Detectron, per le quali saranno mostrati alcuni risultati esplicativi di applicazione.

5.2 TensorFlow: classificazione

TensorFlow sarà il primo framework che sarà testato e sarà quello su cui si incentrerà la fase di testing. Per il task di classificazione, su Tensorflow sono state percorse 3 vie in fase di sviluppo:

- Una prima modalità in cui viene ricercata la libreria "custom" relativa alla rete neurale scelta, nella quale vi sia il codice per fare training, tuning o semplicemente inference, della quale richiamare le funzioni necessarie.
- Una seconda modalità basata su TensorfFlow e sulle sue librerie interne.
- Una terza modalità che userà la Api TensorFlow Slim e le librerie che offre.

5.2.1 Classificazione TF con modello architetturale CUSTOM

Questa è la prima delle modalità sopra descritte: è stato reperito il codice di un modello architetturale, prodotto da qualche utente, per poi farne un pruning del superfluo (ad esempio del codice per allenamento o tuning). Subito dopo individuando le funzioni di nostro interesse, è stato scritto uno script che richiamasse tali funzioni corredando il tutto con funzioni di input e stampa.
Tra le opzioni scelte questa è la via meno consigliata da percorrere, perchè comporta o la scrittura di un codice complesso per il quale spesso è necessario andare a conoscere dei dettagli della fase di training (è consigliato nel caso sia l'utente stesso ad allenare la rete) o comunque è necessario reperire del codice di altri utenti e questo spesso comporta difficoltà in caso di codice poco chiaro o poco commentato.
In primo luogo, sarà fondamentale importare numpy, matplotlib e ovviamente lo stesso Tensorflow, oltre che il codice della rete reperito cioè modified_inception.

```
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import myutils
import modified_Inception
```

In questo specifico esempio è stato usato un modello di rete scaricato dalla rete tramite una URL, che è stato necessario indicare in fase setting insieme ai path di input. Saranno utilizzate delle funzioni "wrapper" di classificazione e predizione tratte da "modified_Inception" mentre in output è stato scelto di fare un pruning sulle prime 10 migliori classi di predizione: spesso in molte applicazioni sarà necessario ridimensionare l'immagine e questo è uno di quei casi specifici visto che il modello inception modificato (modified_inception) lavora su input da 299x299 pixel quindi immagini in formato differente per le quali è necessaria questa operazione di standardizzazione.
Alcune delle operazioni utilizzate per la predizione sono le seguenti:

```
def classify(image_path):
img = Image.open(image_path)
pred = model.classify(image_path=image_path)
model.print_scores(pred=pred, k=10, only_first_name=True)
def plot_resized_image(image_path):
resized_image = model.get_resized_image(image_path=image_path)
plt.imshow(resized_image,interpolation='nearest')
plt.show()
```

Il risultato della predizione per le prime 10 classi su una immagine piccola a qualità ridotta

```
89.11%: giant panda
0.78%: indri
0.30%: lesser panda
0.15%: custard apple
0.12%: earthstar
0.09%: sea urchin
0.05%: forklift
0.05%: digital watch
0.05%: gibbon
0.05%: go-kart
```

Pur avendo utilizzato una rete Inception molto datata si tratta di una immagine molto chiara con un solo soggetto per cui le prime predizioni hanno risultati percentuali decisamente elevati, mentre le predizioni successive sono molto basse: ciò denota uno di quei casi in cui la rete performa bene.

5.2.2 Classificazione con Modello di Rete già allenato

Generalizzando il precedente esempio ed usando un modello di rete pre-allenato, si potrebbe allo stesso modo effettuare una classificazione e quindi etichettare una generica immagine. Ovviamente questo richiede una quantità di codice molto maggiore da dettagliare, codice del quale saranno mostrate soltanto le parti salienti. Sostanzialmente nel main sarà caricato il grafico del modello architetturale e saranno letti anche i tensori ottenuti nel training:

```
graph= load_graph(model_file)
tensor=
read_tensor_from_image_file(file_name,input_height=input_height,
input_width=input_width,input_mean=input_mean,\\
input_std=input_std)
input_name = import/'' + input_layer
output_name = import/'' + output_layer
input_operation = graph.get_operation_by_name(input_name)
output_operation = graph.get_operation_by_name(output_name)
with tf.Session(graph=graph) as sess:
    results = sess.run(output_operation.outputs[0],
                       ({input_operation.outputs[0]: t})
results = np.squeeze(results)
top_k = results.argsort()[-5:][::-1]
    labels = load_labels(label_file)
```

Qui sono importanti le due funzioni load_graph e read_tensor_from_image, e sess.run() oltre alla funzione load_label.
Nel seguente esempio è stata scelta la rete Inception v3 per ottenere le prime 5 predizioni: e' una rete molto più moderna rispetto a quella vista nell'esempio precedente e in teoria dovrebbe essere più performante.
Il risultato di seguito mostrato confermerà ciò, pur trattandosi di una immagine (fig: 5.1) ben definita e allo stesso modo facile da predire

```
92.81%: lorikeet
0.17%: coucal
0.07%: macaw
0.05%: ping-pong ball
0.04%: sulphur-crested cockatoo
```

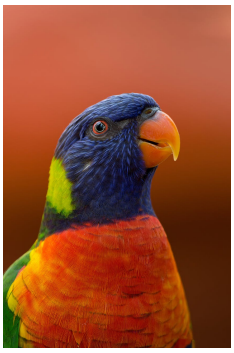


Figura 5.1: Immagine valutata con Inception v3.

5.2.3 Classificazione con TensorFlow Slim

TF Slim, come visto, fornisce molte risorse come reti pre-allenate e librerie grazie ai quali è stato possibile produrre uno script che elaborasse in maniera ottimale le immagini al fine di classificarle. I dati, se già convertiti in TFRecord, sono direttamente utilizzabili altrimenti sarebbe stata necessaria una conversione. Verrà fatto l'import di tutto ciò che servirà (come fatto nei precedenti esempi) tenendo conto che sarà necessario importare anche il "nuovo" Slim, i dataset e ciò che concerne il processamento delle reti, operazioni per le quali viene fornito pieno supporto nel repository.
Di seguito i pezzi di codice di maggiore interesse:

```
from datasets import dataset_utils
from tensorflow.contrib import slim
from datasets import imagenet
from nets import inception
from preprocessing import inception_preprocessing
```

La vera funzione di grosso interesse è quella esposta di seguito: viene preprocessata l'immagine e poi inizializzata la rete con degli appositi parametri, prima di lanciare la predizione a partire da un checkpoint di allenamento, attraverso cui viene fatto partire il processo di inference.

```
image = tf.image.decode_jpeg(image_string, channels=3)
processed_image = inception_preprocessing.preprocess_image(image,
    image_size, image_size, is_training=False) processed_images =
    tf.expand_dims(processed_image, 0)
```

I modelli di rete e i relativi parametri richiedono degli accorgimenti: VGG e ResNet v1 ad esempio richiedono uno shift di 1 del numero delle classi quindi 1000 classi piuttosto che 1001 se viene usato Imagenet come dataset:

```
with slim.arg_scope(inception.inception_v1_arg_scope()):
logits, _ = inception.inception_v1(processed_images, num_classes=1001,
    is_training=False)
```

Il risultato dell'ultimo layer viene dato a un layer softmax:

```
probabilities = tf.nn.softmax(logits)
```

Vengono salvati i parametri nel modello di rete:

```
init_fn = slim.assign_from_checkpoint_fn(
os.path.join(checkpoints_dir, 'inception_v1.ckpt'),
slim.get_model_variables('InceptionV1'))
```

Infine, viene fatta l'inizializzazione della sessione prima di lanciare la funzione che darà in output il vettore delle predizioni:

```
with tf.Session() as sess:
        init_fn(sess)
        np_image, probabilities = sess.run([image, probabilities])
```

È quindi chiaro che conoscendo come utilizzare i vari modelli di rete pre-allenati, con delle semplici modifiche, si possono generalizzare gli script.
Al link [144] sono fornite diverse nozioni da cui potere attingere per compiere le modifiche necessarie per l'uso di vari modelli pre-allenati mentre al link [145] è possibile trovare la lista dei modelli disponibili.

Figura 5.2

Inception V1
Testando questa applicazione su una rete Inception V1 sull'immagine (fig.5.2) si otterranno le seguenti predizioni:

```
45.14%: Cocker spaniel, English cocker spaniel, cocker
21.55%: Sussex spaniel
10.37%: Irish setter, red setter
5.06%: Welsh springer spaniel
2.99%: Clumber, clumber spaniel
```

VGG16
Testandoil tutto su una VGG16 si otterranno i seguenti risultati:

```
86.47%: Cocker spaniel, English cocker spaniel, cocker
5.42%: Sussex spaniel
5.38%: Clumber, clumber spaniel
1.14%: Golden retriever
0.53%: English setter
```

È chiara la differenza di prestazione tra le due reti proposte: una offre predizioni più accurate, una più grossolane.

5.3 TensorFlow: Localizzazione e Classificazione mediante Api Object Detection

In questa sezione si lavorerà sul secondo task di nostro interesse cioè quello di localizzazione unito alla classificazione quindi Object Detection; lo script, prodotto in Python, è editabile come sempre al variare del dataset, del tipo di rete scelta e dei parametri delle immagini di test: tralasciando queste parti di codice sarà importante andare ad analizzare le parti in cui si svolgono le vere e proprie operazioni salienti, come il caricamento del grafo.

```
def load_graphandmodel():
    detection_graph = tf.Graph()
    with detection_graph.as_default():
        od_graph_def = tf.GraphDef()
        with tf.gfile.GFile(PATH_TO_FROZEN_GRAPH, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
        tf.import_graph_def(od_graph_def, name='')
    return detection_graph
```

Il cuore dello script sta in questa funzione da richiamare per eseguire la valutazione di una singola immagine:

```
def run_inference_for_single_image(image, graph):
    with graph.as_default():
        with tf.Session() as sess:
            ops = tf.get_default_graph().get_operations()
            all_tensor_names = {output.name for op in ops for output in
                op.outputs}
            tensor_dict = {}
            for key in ['num_detections', 'detection_boxes',
                'detection_scores','detection_classes',
                'detection_masks']:
                tensor_name = key + ':0'
                if tensor_name in all_tensor_names:
                  tensor_dict[key] =
                      tf.get_default_graph().get_tensor_by_name(tensor_name)
            if 'detection_masks' in tensor_dict:
                detection_boxes =
                    tf.squeeze(tensor_dict['detection_boxes'], [0])
                detection_masks =
                    tf.squeeze(tensor_dict['detection_masks'], [0])
                real_num_detection =
                    tf.cast(tensor_dict['num_detections'][0], tf.int32)
                detection_boxes = tf.slice(detection_boxes, [0, 0],
                    [real_num_detection, -1])
                detection_masks = tf.slice(detection_masks, [0, 0, 0],
                    [real_num_detection, -1, -1])
                detection_masks_reframed =
                    utils_ops.reframe_box_masks_to_image_masks(
                detection_masks, detection_boxes, image.shape[1],
                    image.shape[2])
```

```
            detection_masks_reframed =
                tf.cast(tf.greater(detection_masks_reframed, 0.5),
                tf.uint8)
            tensor_dict['detection_masks'] =
                tf.expand_dims(detection_masks_reframed, 0)
        image_tensor =
            tf.get_default_graph().get_tensor_by_name('image_tensor:0')
        output_dict = sess.run(tensor_dict,feed_dict={image_tensor:
            image})
        output_dict['num_detections'] =
            int(output_dict['num_detections'][0])
        output_dict['detection_classes'] = output_dict[
            'detection_classes'][0].astype(np.uint8)
        output_dict['detection_boxes'] =
            output_dict['detection_boxes'][0]
        output_dict['detection_scores'] =
            output_dict['detection_scores'][0]
        if 'detection_masks' in output_dict:
            output_dict['detection_masks'] =
                output_dict['detection_masks'][0]
    return output_dict
```

Ottenuto il vettore "output_dict" si potranno estrarre i valori relativi a classificazione, box e localizzazione del box. **La predizione ottenuta e la conseguente localizzazione avranno percentuali elevatissime.**

Figura 5.3: Immagine originale.

Figura 5.4: Immagine annotata: la predizione è 94%: Dog per il cane a sinistra, 93%: Dog per il cane a destra.

5.4 Caffe: Classificazione

5.4.1 Classificazione con Rete referenziata da Caffe

Come fatto con Tensorflow, si è provato a produrre uno script che permettesse di testare la classificazione usando il numero maggiore possibile di reti neurali.
La parte iniziale prevede l'import delle solite librerie utili ed indispensabili e del codice per la selezione dei path e la modalità di uso di caffe quindi CPU o GPU; per questo esempio verrà sfruttata una delle reti di riferimento di Caffe, la AlexNet nella versione allenata in maniera differente rispetto alla tradizionale, cioè senza data-augmentation e con la normalizzazione e il pooling invertito.

```
import caffe
net = caffe.Net(model_def,model_weights,caffe.TEST)
```

Una doverosa premessa riguarda gli input: Caffe di default riceve immagini in BGR, quindi verrà utilizzato un transformer nativo cioè "caffe.io.Transformer" per convertirle nel giusto formato oppure un codice autonomo che prenda in considerazione il fatto che Caffe accetta di prendere immagini in formato BGR con valori compresi tra 0 e 255 a cui dovrà essere sottratto un valore del pixel fornito da Imagenet.

```
mu = np.load(caffe_root + 'python/caffe/imagenet/ilsvrc_2012_mean.npy')
mu = mu.mean(1).mean(1)
transformer = caffe.io.Transformer({'data':
    net.blobs['data'].data.shape})
transformer.set_transpose('data', (2,0,1))
transformer.set_mean('data', mu)
transformer.set_raw_scale('data', 255)
transformer.set_channel_swap('data', (2,1,0))
```

In questa sezione del codice verrà prodotto il risultato di predizione.

```
output = net.forward()
output_prob = output['prob'][0]
```

Dalla valutazione della seguente immagine (fig.5.5):

```
9.63% running shoe
8.06% bicycle-built-for-two, tandem bicycle, tandem
7.68% tricycle, trike, velocipede
6.78% unicycle, monocycle
3.30% cradle
```

Figura 5.5: Trattandosi di una immagine poco chiara a livello di contenuto, la predizione non risulta molto accurata o veritiera.

5.5 Caffe: Object Detection

Le R-CNN (Region Based CNN) sono fondamentali per riconoscere più oggetti nella stessa immagine e nel contempo localizzarli: nella cartella models del repository di Caffe, infatti è fornito un modello di R-CNN allenato su ILSVRC13 che è possibile sfruttare per fare un esempio di utilizzo del framework.
Per il nostro scopo servirà anche la ricerca Selettiva, il cui codice è reperibile nel seguente repository:
https://github.com/sergeyk/selective_search_ijcv_with_python
Può essere comodo scaricare il repository e importare il pacchetto e quindi chiamare le funzioni interne per poterla utilizzare direttamente oppure semplicemente si può lanciare l'installazione dei pacchetti come spiegato nella guida fornita.

5.5.1 Ricerca Selettiva

La ricerca selettiva verrà utilizzata per individuare quali sono i box di interesse all'interno di un'immagine da valutare. Di seguito un esempio di immagine (fig.5.6) su cui è applicato l'algoritmo di Ricerca Selettiva, il cui codice è totalmente ripreso dal seguente repository [146], utile a verificare visivamente analizzare come agisce l'algoritmo e capire come si può sfruttare per il nostro task di Object Detection.

5.5.2 Caffe: script di Object Detection

La chiave di questo script sarà la "selectivesearch" e tutto ciò che si collega ad essa, quindi sarà necessario farne un import. Oltre questa saranno necessari altri strumenti di elaborazione grafica uniti agli strumenti che solitamente sono necessari.

```
from skimage import io
import skimage.data
from skimage import img_as_float
import selectivesearch
import dlib
import cv2
import caffe
```

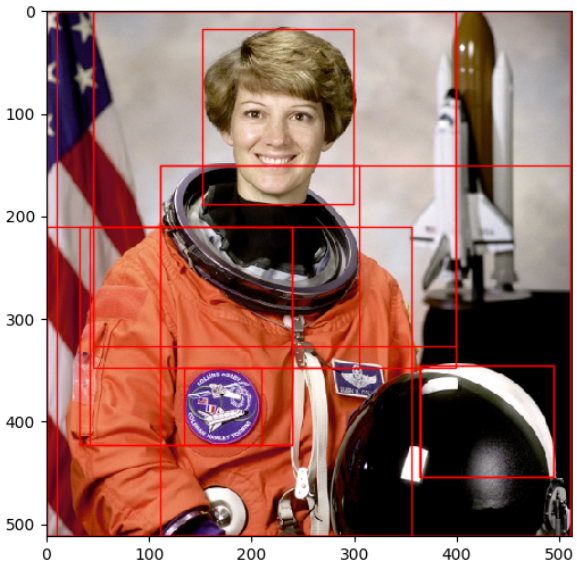


Figura 5.6: Immagine elaborata mediante ricerca selettiva.

In questo caso il modello è allenato su Imagenet per il dataset del contest di classificazione del 2012. Essendo richiesta molta capacità computazionale per la ricerca selettiva è preferibile eseguire lo script in modalità GPU.
Di seguito la parte di codice inerente alla creazione della rete.

```
net = caffe.Net(model_def,model_weights,caffe.TEST)
```

Sarà applicata la trasformazione sull'immagine in BGR visto che il formato di immagine richiesto non è tradizionale:

```
batch_size = 10
mean_img = np.load(caffe_root +
    'python/caffe/imagenet/ilsvrc_2012_mean.npy')
mean_img_avg = np.zeros(3)
mean_img_avg[0] = np.mean(mean_img[0,:,:].ravel())
mean_img_avg[1] = np.mean(mean_img[1,:,:].ravel())
mean_img_avg[2] = np.mean(mean_img[2,:,:].ravel())
transformer = caffe.io.Transformer({'data':
    net.blobs['data'].data.shape})
transformer.set_transpose('data', (2,0,1))
transformer.set_mean('data', mean_img_avg)
transformer.set_channel_swap('data', (2,1,0))
net.blobs['data'].reshape(batch_size,3,227,227)
```

Bisognerà adesso definire le regioni da classificare per poi classificarle ad una per una: è una operazione lunga e complessa che richiede tante linee di codice che non saranno dettagliate, ma che saranno presenti nello script. In particolare, verrà definita una ricerca selettiva ridotta, in cui non verranno prese in considerazioni tutte le box trovate, ma solo quelle più salienti la cui classificazione e dimensione sarà considerabile: la scelta sarà quella di eliminare le regioni di dimensioni inferiori ai 15 pixel.

Saranno usate quindi due funzioni: selective_search() e classify_each_region() che prende l'immagine e le regioni candidate.
Rilevante è anche la funzione boundingbox_analysis(), che analizza i risultati delle due precedenti operazioni cioè le regioni trovate e le loro probabilità di classificazione: verrà ciclato l'output di tutte le regioni e per ciascuna verrà presa la probabilità più alta; se la probabilità è maggiore di un certo valore, designato come 0.4 ad esempio, questo sarà un valore che sarà considerato, altrimenti verrà scartata il box relativo.
Sostanzialmente le chiamate del nostro main saranno le seguenti dove cat_inds è il file che contiene gli indici delle classi:

```
net_output = classify_each_region(img, candidates)
bounding_box = bounding_box_analysis(img, candidates, net_output,
    cat_inds, filename, 1, True)
```

Il test su questa immagine 5.7 produce il risultato seguente:

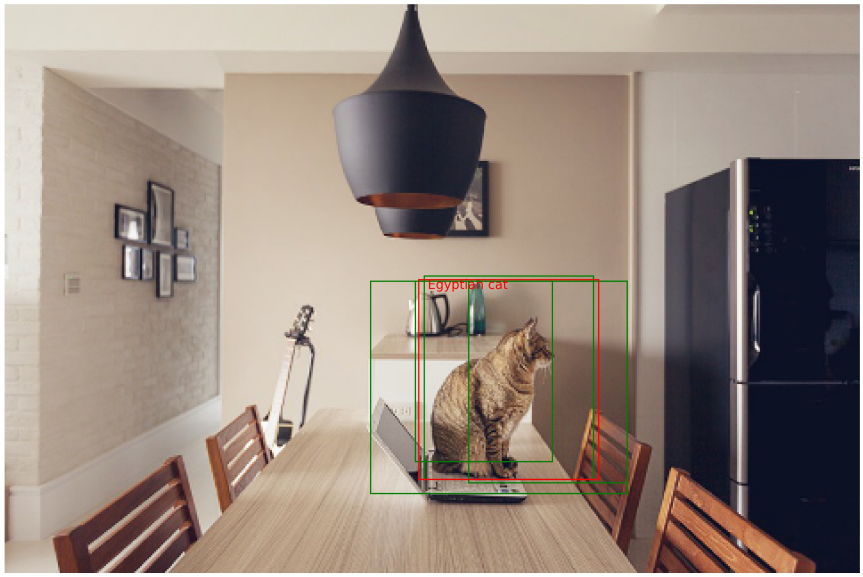


Figura 5.7: Object Detection mediante ricerca selettiva.

5.6 PyTorch

Pytorch è il terzo framework testato: si è cercato di variare il tipo di applicazione, per fornire una vastità di esempi, cercando di orientarsi su implementazioni più complesse. In particolare, si lavorerà con la SSD (Single Shot Detection) e con Yolo per Object Detection e Video Tracking.
Per alcuni di questi script si è scritto del codice che sfruttasse delle chiamate a delle funzioni presenti in delle librerie di alcune Api e quindi codice già prodotto, mentre per altri script si è scritto del codice totalmente autonomo.

5.6.1 PyTorch: Classificazione

Come prima parte di codice vi è il solito passaggio di inclusione delle librerie necessarie: da notare qui l'import di torch e torchvision.

```
from torch import nn
from torch import optim
import torch.nn.functional as F
from torchvision import datasets, transforms, models
```

Se si volesse utilizzare il modello già predefinito dal framework si userà:

```
model = models.densenet161(pretrained=True)
```

Invece volendo utilizzare un modello di pretrainedmodels, già descritto in precedenza, è necessario **import pretrainedmodels**:

```
model_name = 'densenet201'
model = pretrainedmodels.__dict__[model_name](num_classes=1000,
    pretrained='imagenet')
```

Se non si volesse usare il modello, nella versione proposta dalla comunità di PyTorch, ma si volesse utilizzare un altro modello da noi allenato servirà il modello pretrained bloccato ad un determinato checkpoint e il codice della classe "TheModelClass(nn.Module)" che va generato in fase di train.

```
pathmodel= /model/*%*.pth
checkpoint = torch.load(pathmodel'')
model = ThemodelClass(num_classes=1000)
model.load_state_dict(checkpoint)
```

Diversi modelli di rete sono reperibili qui:
https://github.com/pytorch/vision/tree/master/torchvision/models
È obbligatoria di base la definizione della funzione di processamento dell'immagine, nella quale vengono applicate delle funzioni di trasformazione che sono state già eseguite durante la fase di training: Resize che ridimensiona l'immagine, ToTensor che converte l'immagine in formato utilizzabile da PyTorch e Normalize che porta i pixel in un range tra -1 e +1.

```
def processing_image(image_path):
   image_original = Image.open(image_path)
   img_transformation = transforms.Compose([
   transforms.Resize(256),
   transforms.CenterCrop(224),
```

```
transforms.ToTensor(),
transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224,
    0.225]) ])
img = image_transformation(image_original)
```

L'altra funzione fondamentale è quella di predizione dove avviene la scelta dell'uso della CPU piuttosto che della gpu e il calcolo della predizione dell'immagine.

```
def predict(image_path, model):
   model.eval()
   model.cpu()
   img_final = processing_image(image_path)
   img_final = img_final.unsqueeze_(0)
   img_final = img_final.float()
   output=model.forward(img_final)
```

L'output ottenuto sarà ritornato e sfruttato per creare la conversione da array di tensori ad array numpy e poi per la conversione in label vere e proprie.
Sull' immagine in fig.5.8 si avrà il seguente risultato per le prime 5 predizioni, con l'uso di un modello di rete Densenet.

```
79.63% cardoon
17.84% artichoke, globe artichoke
0.0030% half track
0.0027% coral fungus
0.0023% earthstar
```

Figura 5.8: Immagine classificata con script Python con rete DenseNet: il risulto ottenuto è molto performante per via dell'immagine ben definita come contenuto e per l'uso di una rete molto accurata.

5.6.2 PyTorch: SSD Single Shot Multibox Detector[70]

SSD è uno degli algoritmi più popolari per Object Detection e risulta essere veloce come il Faster R-CNN. I modelli più vecchi si sviluppavano in due parti implementative: una prima parte in cui viene proposta una regione da classificare e un seguente classificatore per la zona proposta. Questo era computazionalmente costoso e complicato per le applicazioni real-time.
Il multibox invece è una tecnica che predice un bounding box come un problema di regressione e per ogni box vengono generati punteggi per vari oggetti localizzati. Il bounding-box a livello implementativo è un semplice box, che può contenere diversi oggetti, e che è identificato da delle coordinate.
SSD invece è una pura architettura convolutiva organizzata in 3 parti: Convoluzione Base, Convoluzione Ausiliaria e Convoluzione di Predizione.
La convoluzione base è derivata da una esistente architettura di classificazione che propone una feature map di convoluzione di basso livello; la convoluzione ausiliare invece aggiunge una convoluzione di livello superiore e infine la convoluzione di predizione va identificare e localizzare gli oggetti in queste mappe.
È stato necessario reperire online i modelli che solitamente vengono usati, chiamati SSD300 e SSD 512, sui quali rispettivamente si possono usare immagini con dimensione 300x300 in formato RGB e immagini più grandi sulla seconda. È venuta fuori una rete più grande che a livello teorico dovrebbe fornire prestazioni migliori.
Verrà presentato un esempio di classificazione e localizzazione, partendo da un checkpoint di rete pre-allenata e verranno riassunti i risultati ottenuti. Ovviamente anche in questo esempio è necessario un file di label per categorizzare gli elementi e anche la stessa immagine di input necessita di essere preprocessata per essere fruibile automaticamente dal modello di architettura utilizzato.
Documentazione aggiuntiva, dettagli implementativi e informazioni sui modelli e sulle funzioni di utilizzo sfruttate nel nostro script sono ritrovabili nel seguente repository:
https://github.com/sgrvinod/a-PyTorch-Tutorial-to-Object-Detection

Il nostro esempio sfrutterà un SSD300, per il quale miglior checkpoint dello sviluppatore che ha allenato il modello su TitanX Pascal, è alla "epoch" 186 con una validation loss di 2.515.

```
if torch.cuda.is_available():
   checkpoint = torch.load(checkpoint)
else:
    checkpoint=torch.load(checkpoint,device)
start_epoch = checkpoint['epoch'] + 1
best_loss = checkpoint['best_loss']
model = checkpoint['model']
model = model.to(device)
model.eval()
```

Sarà convertita l'immagine in modo da renderla consona all'input, settando 300x300 come risoluzione.

```
resize = transforms.Resize((300, 300))
to_tensor = transforms.ToTensor()
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225])
```

La funzione applicativa vera e propria è la seguente: riceve l'immagine originale, lo score minimo per ciascuna classe, la sovrapposizione massima che devono avere due box, i top k che devo considerare, e 'none' in riferimento al fatto che sia poco sicura una classe e ritorna una immagine annotata; se non viene riconosciuta nessuna etichetta verrà ritornata l'immagine originale e l'etichetta verrà segnalata con '0'.

```
def detect(original_image, min_score, max_overlap,
    top_k,suppress=None):
   image = normalize(to_tensor(resize(original_image)))
   image = image.to(device)
   predicted_locs, predicted_scores = model(image.unsqueeze(0))
   det_boxes, det_labels, det_scores =
       model.detect_objects(predicted_locs,predicted_scores,
   min_score=min_score,max_overlap=max_overlap,top_k=top_k)
   det_boxes = det_boxes[0].to('cpu')
   original_dims = torch.FloatTensor([original_image.width,
       original_image.height,
   original_image.width,original_image.height]).unsqueeze(0)
   det_boxes = det_boxes * original_dims
   det_labels = [rev_label_map[l] for l in
       det_labels[0].to('cpu').tolist()]
   if det_labels == ['background']:
       return original_image
```

Ritornata l'immagine sarà compito dello script la gestione del box, a livello estetico e il font della classificazione. Dal main verrà lanciata la funzione appena definita con dei parametri a piacimento.

```
detect(original_image, min_score=0.2, max_overlap=0.5,
    top_k=200).show()
```

I risultati ottenuti testando questo script sulla seguente immagine 5.9 sono i seguenti:

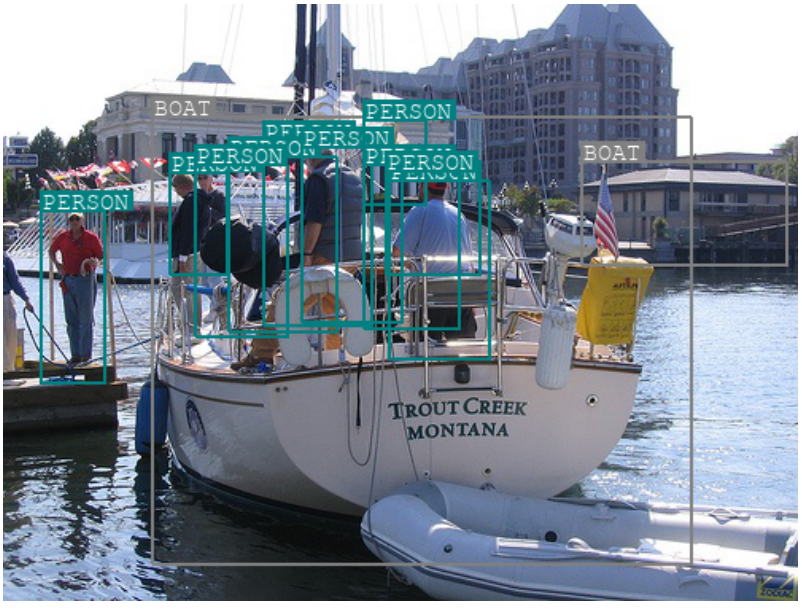


Figura 5.9: Object detection con SSD.

5.6.3 PyTorch: Yolo per Object Detection

Yolo è uno degli strumenti più performanti in termini di Object Detection ma è anche un ottimo modello architetturale ibrido misto tra una rete e una Api usato anche per tracciare degli oggetti in un video, implementazione che sarà dettagliata in questo capitolo. In questo esempio invece sarà utilizzato Yolo attraverso PyTorch, sfruttando l'ultima versione pre-allenata del modello cioè Yolo v3 reperibile sul sito ufficiale:
https://pjreddie.com/media/files/yolov3.weights
allenato sul dataset Coco e sfrutta la DarkNet.
Molte delle funzioni utilizzate sono state sviluppate in questo repository:
https://github.com/cfotache/pytorch_objectdetecttrack
nella quale sono proposte tutte le librerie necessarie.
Le parti di codici più salienti sono le seguenti:

```
config_path='model/yolov3.cfg'
weights_path='model/yolov3.weights'
class_path='model/coco.names'
img_size=416
conf_thres=0.8
nms_thres=0.4
model = Darknet(config_path, img_size=img_size)
model.load_weights(weights_path)
model = model.to(device)
model.eval()
classes = load_classes(class_path)
if torch.cuda.is_available():
   Tensor = torch.cuda.FloatTensor
else:
   Tensor=torch.FloatTensor
```

La funzione di predizione che sarà chiamata dal main si occuperà anche della trasformazione:

```
def detect_image(img):
   ratio = min(img_size/img.size[0], img_size/img.size[1])
   imw = round(img.size[0] * ratio)
   imh = round(img.size[1] * ratio)
   img_transforms = transforms.Compose([ transforms.Resize((imh, imw)),
    transforms.Pad((max(int((imh-imw)/2),0),
       max(int((imw-imh)/2),0), max(int((imh-imw)/2),0),
       max(int((imw-imh)/2),0)),(128,128,128)),
   transforms.ToTensor(),])
   image_tensor = img_transforms(img).float()
   image_tensor = image_tensor.unsqueeze_(0)
   input_img = Variable(image_tensor.type(Tensor))
   with torch.no_grad():
       detections = model(input_img)
       detections = non_max_suppression(detections, 80, conf_thres,
           nms_thres)
    return detections[0]
```

Dal main infine sarà chiamata una funzione che si occupa della definizione grafica dei box e del calcolo delle relative coordinate. I risultati ottenuti sull'immagine 5.10 sono i seguenti:

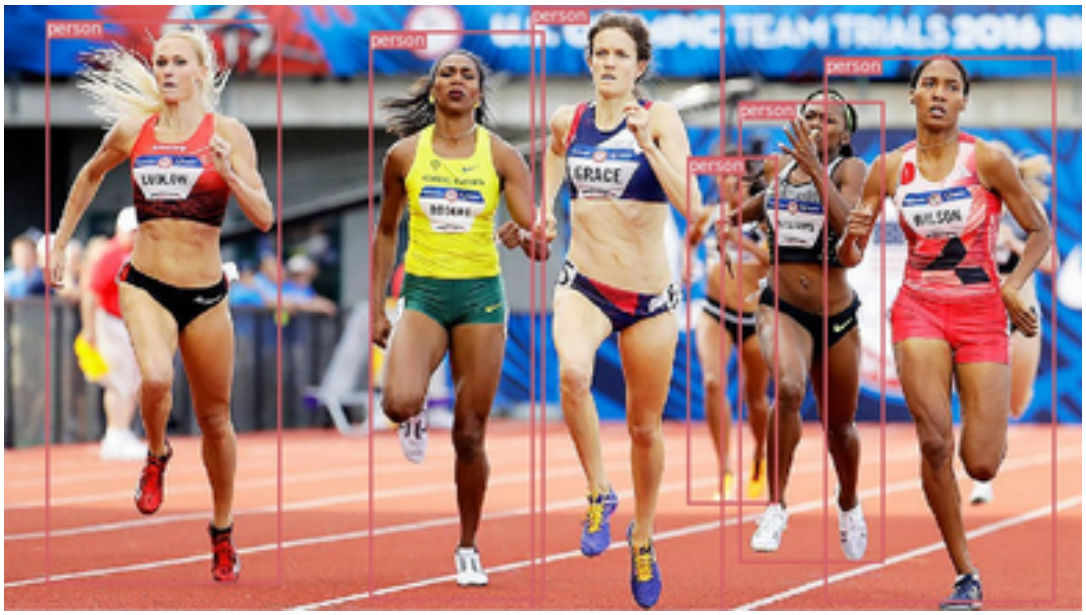


Figura 5.10

5.6.4 Pytorch: Yolo per Video Object Tracking

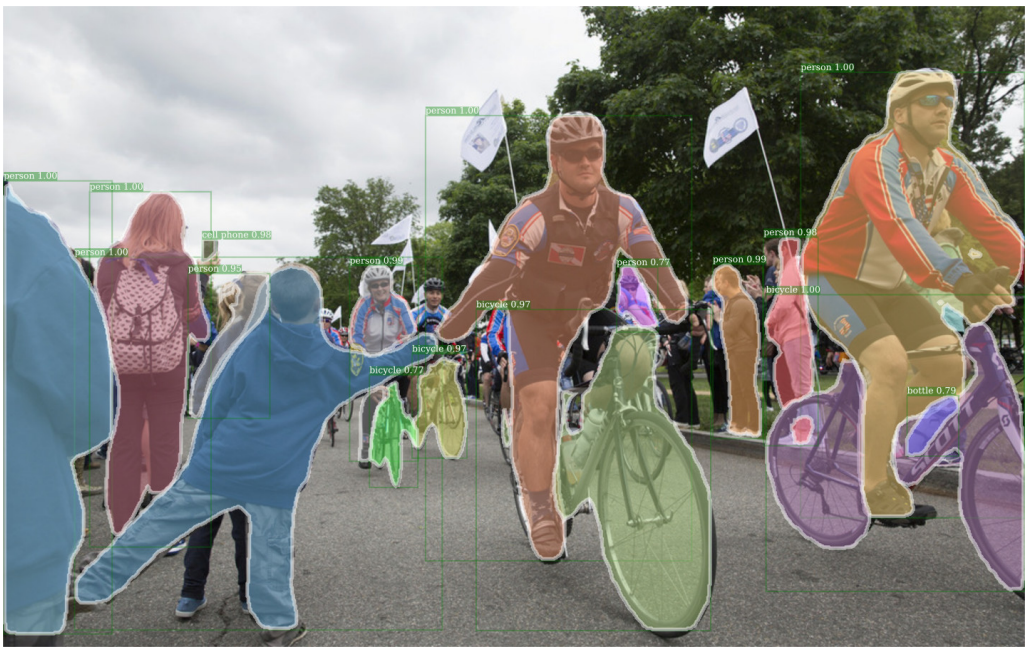
La stessa libreria utilizzata nell'esempio precedente fornisce la possibilità di tracciare gli oggetti all'interno di una immagine.
È chiaro che si tratta di due task totalmente diversi: la localizzazione di un soggetto/oggetto all'interno dei frame che compongono un video è un buon esempio, però diventa complicata nel momento in cui durante il video appaiono frame con più elementi da localizzare. **MOT (Multiple Object Tracking** [147] [148]
Il repository
https://github.com/cfotache/pytorch_objectdetecttrack
fornisce un esempio di questa implementazione che sfrutta la localizzazione multipla di oggetti specifici nel tempo: tutto ciò viene implementato con diversi algoritmi ma tra questi quello che risulta essere più veloce è il SORT (Simple Online and RealTime Tracking) [149], proposto nel 2017 da Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, Ben Upcroft grazie all'uso di un filtro di Kalman [150].
In particolare, Alex Bewley ha scritto una implementazione in Python che verrà utilizzata nel nostro progetto. In questa implementazione è stato usato Yolo v3 per riconoscere i singoli oggetti, allenato su dataset Coco mentre per la lettura del video e la rappresentazione dei frame video si userà **OpenCV**, che sarà importato come cv2 insieme al pacchetto dell'algoritmo SORT.

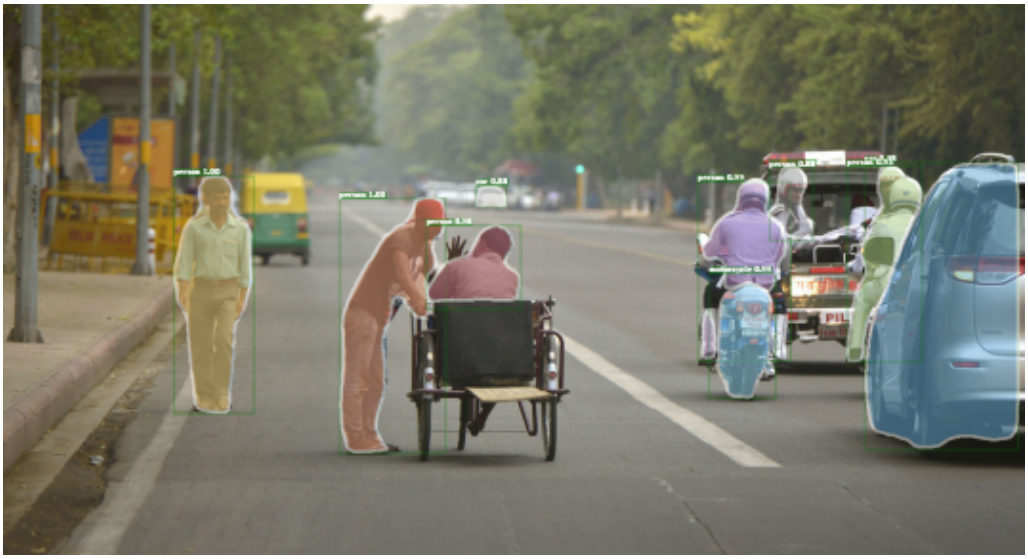
```
import cv2
from IPython.display import clear_output
from sort import *
```

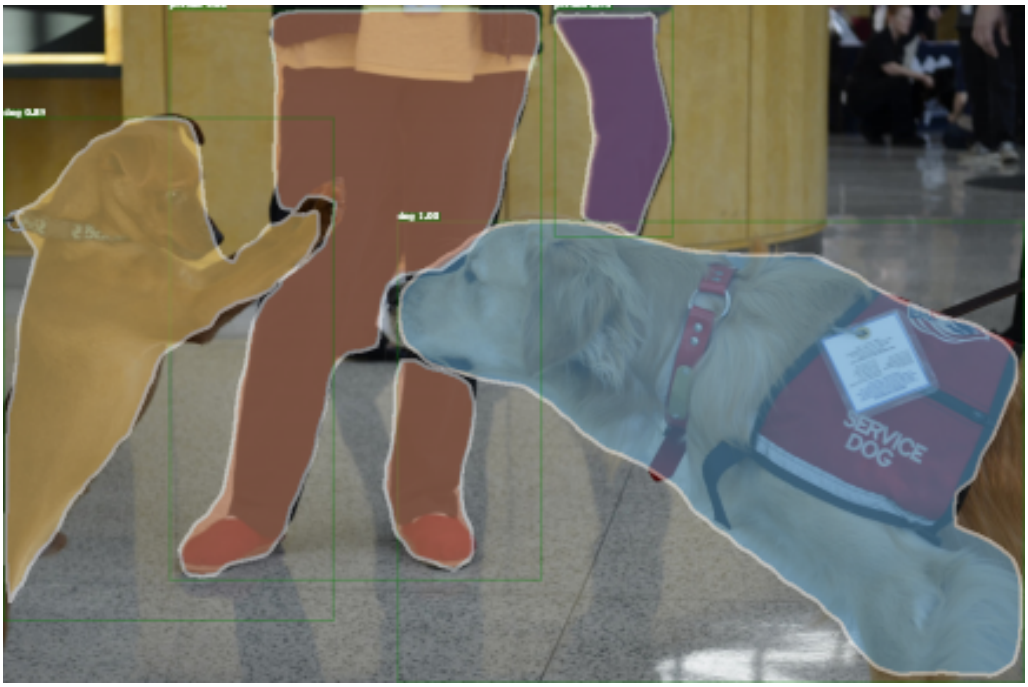
La restante parte di codice è reperibile nel repository linkato ed è piuttosto lunga ed articolata, per cui è stata semplicemente testata.
Per ovvi motivi non sarà possibile dare un esempio visivo ma si daranno dei dati matematici sulla computazione: si tratta di una operazione dispendiosa in termini di calcoli e tempi, infatti facendo un testing su un video di 42 secondi di dimensione 1280x720 composto da 1241 frames, si ottiene un video etichettato in 778.37 secondi, con velocità di esecuzione pari a 0.59 frame al secondo.

5.7 Semantic Segmentation e Detectron

Partendo dall'ampia descrizione fatta in merito a questo task, dall'ampia descrizione fatta per questa API nel capitolo precedente e sapendo che permette di testare alcuni tra gli ultimi modelli più performanti di reti neurali, si è cercato di reperire degli esempi demo pronti e testati per toccare con mano le vere potenzialità dell'applicazione in oggetto. Effettivamente attraverso **Detectron** si è potuto testare ciascuno dei tre task analizzati in questa tesi. All'interno di https://github.com/facebookresearch/Detectron viene proposto un esempio già predisposto per valutare una immagine con una MASK R-CNN ed una rete FPN allenata su dataset Coco 2014, anche se in realtà per effettuare altri esempi basterà variare i parametri in base alla disponibilità dei modelli di rete forniti nel repository indicato che consta di svariati documenti ed informazioni dettagliate per il lancio dello script in maniera opportuna e a nostro piacimento.
È stato testato questo software proprio perchè ci ha permesso di valutare anche il task della Segmentazione Semantica degli oggetti quindi la contornazione dei soggetti rappresentati nelle immagini attraverso le MASK R-CNN.
L'operazione di segmentazione è molto costosa e richiede molte risorse, quindi è preferibile testarla con l'ausilio di una GPU.
Alcuni esempi di risultati ottenuti per la semantic segmentation sono i seguenti:






SERVICE
DOG

Capitolo 6

Test Qualitativi e di Robustezza

6.1 Introduzione alla fase di testing

Nei capitoli precedenti è stato esposto dettagliatamente sia il contesto storico che l'ambito tecnico-matematico ed applicativo in merito al Image Recogniton. In questa sezione saranno analizzati i dati ottenuti dal processamento di una serie di immagini con gli applicativi visti. Sono state apprese diverse informazioni e sfaccettature tecniche, che non erano totalmente chiare nella letteratura teorica ma soprattutto sono stati individuati svariati comportamenti tenuti in determinate situazioni, evidenziando una serie di **"pattern comportamentali"**. Tutto ciò in unione ai dati acquisiti e ai grafici estrapolati permetterà di formulare alcune semplici ma importanti considerazioni sulla robustezza delle reti trattate. Per completezza di esposizione, si noti che il "primo task" si è occupato di performare una semplice classificazione dell'oggetto principale presente nell'immagine mentre il "secondo task" ha portato a localizzare e subito dopo classificare uno o più oggetti individuabili nell'immagine.

6.2 Scelta delle Reti Neurali e degli Applicativi

La fase di testing, esposta di seguito, presuppone una fase preliminare di scelte tecniche che andranno poi ad influenzare in maniera impattante la nostra ricerca: la scelta del modello di rete neurale da utilizzare per ciascuno dei due task, il numero di immagini su cui lavorare e quindi di conseguenza da che dataset attingere per la selezione delle immagini, la scelta del "file di label" per la definizione di tutte le possibili classi e non ultima la scelta di uno o due applicativi tra quelli sviluppati che meglio sposassero i nostri interessi, saranno degli elementi chiavi per il raggiungimento del nostro scopo finale.
Partendo proprio dalla scelta dell'applicativo, si può dire che non è si è trattato di un compito semplice disponendo di tre framework (Tensorflow, Pytorch e Caffe), tutti e tre ampiamente analizzati nel dettaglio nei capitoli precedenti. Si è deciso di convergere su **Tensorflow** che ha permesso di portare avanti il testing su entrambi i task: nello specifico è stato scelto perchè fornisce delle Api con delle librerie già predisposte, quali **TF Slim** e **TF Object Detection**. Queste, come visto, forniscono ampio supporto per l'utilizzo di una grande varietà di reti dalle più datate alle più moderne attraverso le quali, sfruttando determinate chiamate a funzioni e variando poche e mirate righe di codice al variare della rete o del dataset, si può avere uno script applicativo facilmente adattabile a ogni tipo di testing. Quindi la prima scelta è stata fatta scegliendo un criterio di "flessibilità".

La seconda scelta è stata quella del dataset: il più utile è apparso il **Dataset COCO**, per il quale è stato usato anche il relativo "file di label", fornito sia nel package stesso ma anche nel package della rete neurale che chiaramente era allenata su questo. La sezione del dataset utilizzata è quella di testing che consta di 5000 mila immagini, sulle quali è stata fatta una selezione accurata e manuale dei contenuti, proprio per indirizzare la ricerca verso quei pattern comportamentali di cui si è fatto cenno in precedenza. La scelta oculata delle immagini è stato uno degli elementi intuitivi che ha fornito un rilevante aiuto nell'elaborazione dei dati.
Un'altra scelta, ultima in ordine di elenco ma prima in ordine di importanza riguarda l'elemento che riveste il ruolo principale all'interno del nostro contesto cioè la rete neurale. Avendo analizzato e dettagliato diverse famiglie di reti e per ciascuna catalogato i numerosi modelli, vi erano svariate possibilità ma si è deciso di convergere verso una rete che fosse stata sviluppata ed allenata per essere applicata ad entrambi i task ma soprattutto una rete che desse anche un buon compromesso in termini di computazione e performance di predizione: la **Inception ResNet v2**.
Questa godeva già di una buona fama letteraria riguardo alla propria "capacità" e "precisione predittiva", elemento che sarà confermato numericamente dai valori predittivi ottenuti. In particolare, però per il secondo task è stata impiegata sfruttando la tecnica denominata **Faster CNN**.
È stato necessario optare per ulteriori scelte molto più dettagliate in seno alle operazioni di predizione: ad esempio il taglio alle prime 5 predizioni per la classificazione, visto e considerato che con l'uso di reti molto performanti la prima predizione si assesta generalmente su percentuali molto alte e già a partire dalla terza la percentuale spesso tocca valori sotto l'1%; altro elemento di pruning è stato il considerare, nel secondo task, un numero massimo di oggetti localizzati in una immagine facendo un taglio dei box che avessero percentuale di predizione inferiore ad un determinato valore soglia: il tutto per evitare di incappare in falsi positivi o in predizioni senza un vero senso realmente valido ai fini della ricerca (fig. 6.1).

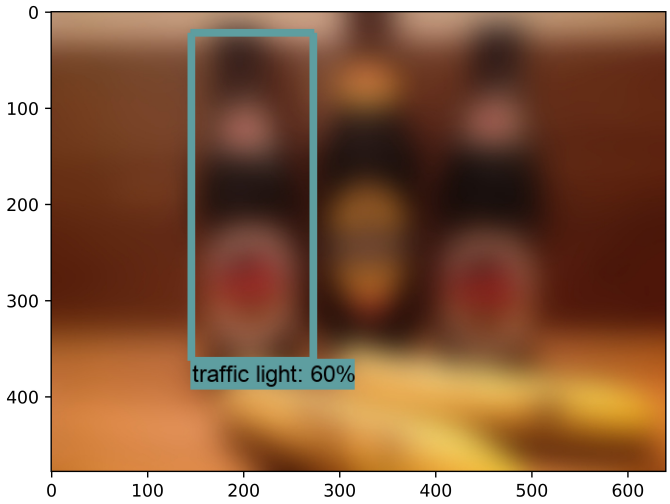


Figura 6.1: Nell'immagine, in figura, che ha subito una distorsione Blur, vi è una bottiglia ma la localizzazione indica un semaforo con predizione del 60% ad un PSNR=18

Il punto di rottura

Un altro punto saliente della fase di testing è stato **il punto di rottura**. È un elemento che è stato individuato durante l'elaborazione dei dati e anche il nome che gli è stato affibbiato è una scelta puramente arbitraria. Corrisponde esattamente al punto in cui la rete inizia a fallire nella predizione e non si tratta di un punto scientificamente esatto ma di una stima del momento in cui la rete inizia a fornire una predizione in output sbagliata o in fase discendente in termini di precisione percentuale, giungendo ad un valore tale da essere inferiore alla seconda predizione. È quindi il punto in cui la rete non ha più predizioni ottimali e "teoricamente" inizia a fornire classificazioni errate.
L'utilizzo di reti molto accurate fornisce un aiuto concreto nell'individuazione di tale punto, poichè queste restano per lunghi tratti sono costanti nelle predizioni ma nel momento in cui vi è un punto di rottura questo è facilmente individuabile anche dall'osservazione della relativa curva delle percentuali. Lavorando su batch di immagini numerose si è preferito sviluppare dei semplici script che individuassero il punto di rottura a partire dagli output forniti dalle elaborazioni della rete neurale.
Il punto di rottura, così analizzato, appare come qualcosa di poco tangibile: per avere maggiore chiarezza diventa necessario introdurre la modalità in cui è stata portata avanti la nostra ricerca. Partendo da delle immagini originali, tratte dal dataset, è stata ridotta la qualità visiva della foto applicando gradualmente delle distorsioni via via crescenti. Ciò fa sì che la rete inizi a faticare nell'analisi di una immagine, e le percentuali di predizione inizino a ridursi drasticamente. Ovviamente aumentando la distorsione di una immagine in maniera sensibile il punto di rottura si presenta in maniera sempre più evidente, poichè si giunge al punto in cui le immagini diventano irriconoscibili anche all'occhio umano.
Per convenzione, sui grafici, la notazione di confronto tra la qualità delle varie immagini sarà il PSNR, che sarà analizzato nelle seguenti sezioni, nonostante per ciascuna delle distorsioni scelte siano stati applicativi diversi livelli crescenti di applicazione della distorsione, che per nostra scelta saranno denominati incrementi σ. È stata testata anche la codifica SSIM, che pur utilizzando una notazione diversa, nell'ambito della nostra riceca fornisce risultati similari.
Le distorsioni applicate in questa fase saranno il **Blur**, la **variazione di Contrasto**, il **Noise** e la **riduzione della qualità JPEG**. Ad onor del vero, si era provato ad utilizzare anche una variazione sull'immagine basata sull'estrazione dei contorni dei soggetti, chiamata Edge, però i grafici ottenuti fornivano dei valori impredicibili e totalmente aleatori. Tra le 4 citate, la variazione di Contrasto e la riduzione di qualità non sono propriamente delle distorsioni a tutti gli effetti, quindi è opportuno per chiarezza descrittiva sottolineare che verranno indicate come "distorsioni" per comodità di esposizione.
Prima di andare a definire nel dettaglio le suddette distorsioni, diventa fondamentale introdurre le già citate notazioni che saranno utilizzate quando si parlerà dei grafici ottenuti: il **PSNR**, che essendo standardizzato permetterà di avere una stima scientificamente valida in merito alla qualità delle immagini; in altre circostanze come anticipato si farà anche riferimento ai livelli di applicazione della distorsione all'immagine: è stato definito un punto di origine corrispondente alla qualità originale e diversi livelli contraddistinti dal valore σ di applicazione. Ad esempio, nel Blur la qualità originale sarà la zero, così come nel Contrasto e nel Noise mentre per la riduzione di qualità bisogna fare un discorso a parte dal momento che il valore di origine pari alla qualità nativa dell'immagine sarà 100. Gli incrementi σ applicati possono arrivare fino ad un massimo di σ =20 per il Blur e di σ =100 per Contrasto e Noise, mentre per la riduzione di qualità JPEG nuovamente il comportamento è un po' diverso poichè con riduzioni via via progressive si giunge al valore finale di qualità minima pari a 1.

PSNR

Il PSNR è una misura adottata per valutare la qualità di una immagine compressa rispetto all'originale ed in quanto tale può essere sfruttato come indice di qualità delle immagini. Dal punto di vista matematico è definito come il rapporto tra la massima potenza di un segnale e la potenza di rumore che può rendere non fedele la rappresentazione del segnale stesso [151]. Viene utilizzato per le immagini che hanno delle compressioni lossy quindi in cui vi è perdita di qualità. In questa ricerca è stato calcolato a partire dall'immagine originale rispetto a quelle distorte. All'aumentare del PSNR si avrà una somiglianza sempre maggiore dell'immagine distorta rispetto a quella originale, quindi si avranno a PSNR più alte immagini tendenti all'originale dal punto di vista della percezione ottica.
Le operazioni pratiche per giungere al calcolo finale del PSNR hanno previsto la conversione sia delle immagini originali che quelle modificate in formato YUV. Il termine YUV, prima battezzato YcrCb, è stato un modello di rappresentazione per la codifica di immagini e video, concepito per rispecchiare il comportamento della visione umana dato che per l'uomo la componente della luminanza è considerata la più importante [152]. Da non scambiare, con il termine Y'UV (YpbPr) che si considera come la divisione dello spazio colore in 3 componenti che veniva usata per trasmissioni video in formato analogico facendo transitare su cavi diversi le informazioni cioè la luminosità detta luminanza Y, cioè l'informazione in bianco e nero, e le due componenti di crominanza cioè i colori Pb e Pr.
La codifica qui utilizzata invece è quella del fenomeno del sottocampionamento della crominanza [153], nella quale si dimezza la risoluzione del colore secondo lo schema 4:x:x e nello specifico 4:2:2. Questo risulta essere lo schema più comune dato che solo i sistemi ad alta gamma, a tutt'oggi, sono in grado di campionare le informazioni di luminanza e crominanza con la stessa risoluzione cioè 4:4:4.
Il calcolo del PSNR è stato permesso dal tool **VQMT: Video Quality Measurement Tool** [154], che normalmente opera sui video e nello specifico su singoli frame che non sono altro che singole immagini. Per completezza, di seguito, le righe di codice generiche utilizzabili per il calcolo del PSNR per una singola immagine attraverso il tool sopra citato.

```
magick convert test_image_$a/image_OUTPUT$X.jpg
   test_finalyuv_$a/image_OUTPUT$X.yuv
 vqmt test_original_processed_yuv/image_OUTPUT$X.yuv
    test_finalyuv_$a/image_OUTPUT$X.yuv 200 200 1 1
    ./result$a/result$X PSNR
```

Nei seguenti paragrafi pertanto si farà riferimento a questa misura "standard".

6.3 Gestione della Fase di Testing

I due filoni della fase di testing cioè **Classificazione** e **Object Detection** saranno divisi a loro volta in due fasi operative per ciascuno: una prima fase in cui si è lavorato su un numero ridotto di immagini pari a 10 unità, che ha permesso di focalizzarsi su un'analisi più accurata e precisa dei risultati ottenuti ed una seconda fase definibile come "massiva" in cui si è lavorato su una quantità di immagini più ampia che ha permesso di studiare il comportamento della rete su larga scala. La classificazione essendo meno costosa a livello computazionale, per quanto riguarda le operazioni di predizione nel caso dei test massivi, è stata portata avanti su 250 immagini, mentre per la localizzazione il batch (sempre per la fase massiva) era composto da 50 immagini.
Per la prima fase invece, partendo dal dataset originale, sono state selezionate 10 immagini con criteri specifici: immagini con un oggetto/soggetto principale in primo piano che fosse l'unico elemento nella foto, in modo tale da ottenere una predizione sull'immagine originale definita e precisa. Tutto ciò per avvicinare la fase di testing a comportamenti tendenzialmente reali e paragonabili all'occhio umano.
Questo passo aggiuntivo di selezione nasce da un'osservazione sulle possibili applicazioni reali che può avere il nostro studio: navigando sul web infatti in diversi repository online iniziano ad essere presentate delle applicazioni real-time, machine learning based, che si occupano di produrre una classificazione/localizzazione ricevendo in input immagini ottenute sul momento con Webcam o fotocamere ma anche dal fatto che mediante il riconoscimento immediato di oggetti, molti siti/app di E-commerce permettono di trovare esattamente l'oggetto ricercato. Tali evidenze hanno spinto a portare avanti dei test che dessero un minimo di riscontro reale.
Alla luce di ciò, per mostrare anche le debolezze e le difficoltà della nostra rete, nella selezione sono state aggiunte anche delle immagini incerte, la cui poca chiarezza fosse data dal fatto che l'oggetto fornisse un labelling multiplo: ad esempio un gatto o un cane possono essere catalogati con una label generica (quindi "dog", "cat") o con una label più specifica relativa magari alla razza di ciascuno. Questo creerà delle anomalie visibili nelle rispettive curve di predizione di tali immagini.
Nella seconda fase invece si è lavorato con una quantità di immagini pari a 250, immagini totalmente casuali quindi senza criterio accurato di selezione: lo studio si è qui spostato su un discorso per lo più quantitativo che qualitativo.
Per quanto riguarda il task di localizzazione, il passo aggiuntivo di selezione delle immagini, è stato fatto cercando di individuare invece immagini con più oggetti possibili o immagini con più soggetti chiaramente riconoscibili.
I dati ottenuti dall'elaborazione di immagini con la nostra rete sono stati rappresentati attraverso dei grafici in grado di evidenziare comportamenti più o meno classificabili: a partire dal primo task saranno date delle evidenze prima su delle singole immagini per poi passare ad evidenziare i comportamenti su più immagini.

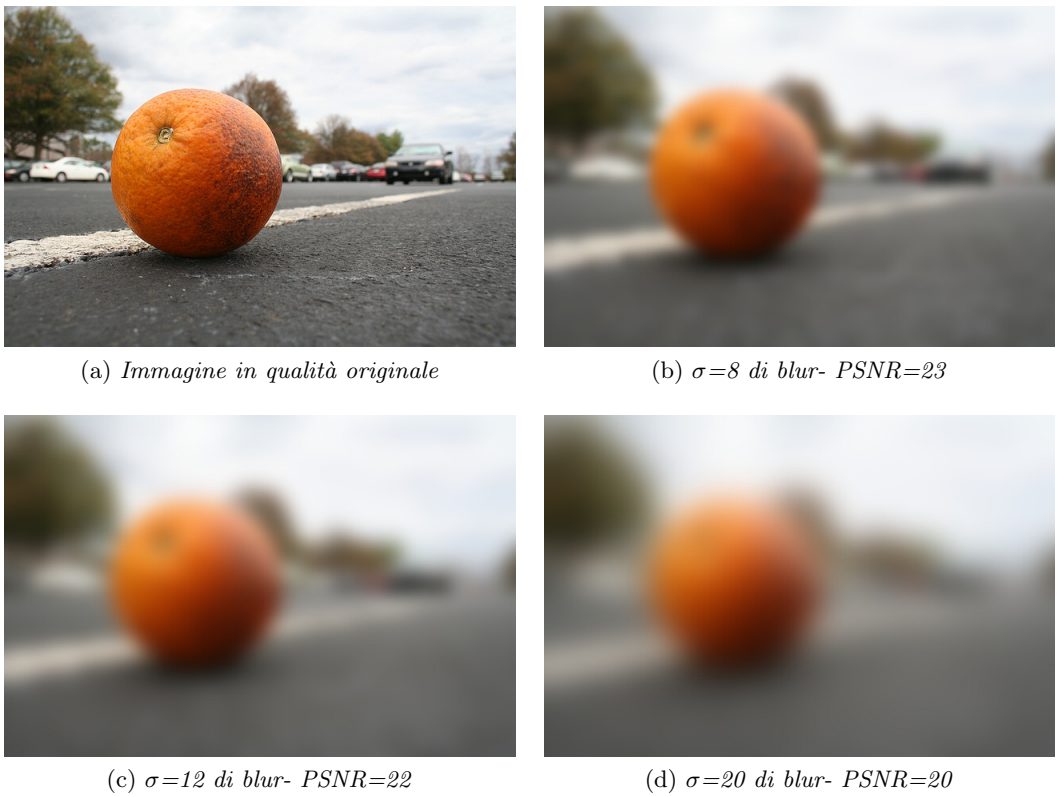
Blur

Il **Blur** è il risultato di una sfocatura sull'immagine attraverso l'uso di una funzione di **Convoluzione Gaussiana** o di una **Convoluzione tradizionale**. Oggi risulta essere una delle tecniche più utilizzate per creare ombre nelle immagini e proprio per questo appare paragonabile all'effetto che si crea quando non si mette a fuoco in maniera ottimale un soggetto con una fotocamera. Il dettaglio dell'immagine viene ridotto sensibilmente in base al livello con cui viene applicato il "blurring".

All'interno del tool utilizzato cioè ImageMagick [155] sono disponibili due tipi di operatori molto simili tra loro: -"blur" e -"gaussian-blur". Vi è molta confusione su quale sia migliore in termini applicativi: il blur tradizionale tra i due è più veloce poichè usa una tecnica che opera in due stage su ciascuno dei due assi mentre il gaussian-blur è matematicamente più preciso dal momento che viene applicato in maniera simultanea su entrambi gli assi. Il costo di differenza tra i due è molto elevato, motivo per il quale si è lavorato sul Blur tradizionale (-blur) dovendo performare l'operazione su una quantità ampia di immagini.

```
magick test_image/image1.jpg -blur 0x20 test_image_20/image1proc.jpg
```

Nel comando appena listato 0x20 corrisponde a radius x sigma, dove sigma è l'argomento importante ed è la quantità di blurring che verrà applicato e praticamente la nostra σ mentre il radius è usato per determinare la dimensione dell'array su cui calcolare la distribuzione: questo dovrebbe essere un valore intero ma se viene dichiarato come zero o non viene dichiarato, si può sfruttare per l'operazione il valore del raggio più largo possibile, in modo da fornire dei risultati più significativi per l'applicazione della distribuzione Gaussiana.
Nelle immagini della figura 6.2 si evidenziano 4 esempi di applicazione di blurring: la rete si rompe già nella figura b) dove la predizione diventa Croquet Ball cioè una palla per il Croquet, uno sport americano, mentre nella d) è visibile una applicazione massimale di Blur che riduce quasi al minimo le prestazioni di predizione della rete.



(a) *Immagine in qualità originale*

(b) *σ=8 di blur- PSNR=23*

(c) *σ=12 di blur- PSNR=22*

(d) *σ=20 di blur- PSNR=20*

Figura 6.2: Immagine con applicazione di Blur

Riduzione della Qualità JPEG

La seconda delle "distorsioni", seppur non si tratti propriamente di una distorsione, ma di un peggioramento dell'immagine, è essenzialmente la diminuizione della **qualità** intrinseca dell'immagine fino a renderla praticamente indecifrabile. Nel seguente comando è stato presentato un esempio su come lanciare una modifica dell'immagine per ridurla alla qualità del 30% rispetto al valore massimo.

```
magick test_image/image9.jpg -quality 30 test_image_100/image9proc.jpg
```

(a) *Immagine con qualità dimezzata, non si notano le modifiche*

(b) *Immagine con qualità pari al 15% rispetto all'originale-PSNR=28*

(c) *Immagine con qualità pari al 3% rispetto all'originale-PSNR=23*

(d) *Immagine con qualità pari al 1% rispetto all'originale-PSNR=22*

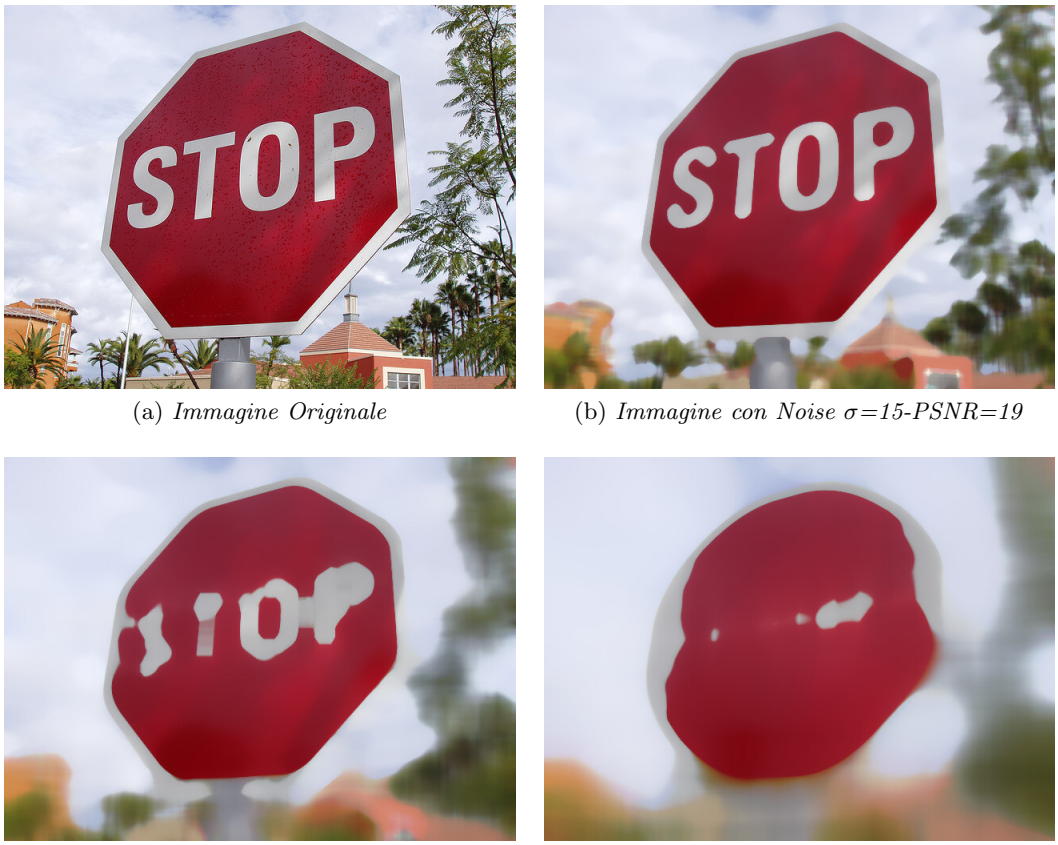
Figura 6.3: Immagine con applicazione della riduzione della qualità JPEG

Nella serie di 4 immagini in figura 6.3 si nota come la riduzione della qualità JPEG delle immagini, pur peggiorandole sensibilmente a vista d'occhio, le renda ancora decifrabili.

Noise

L'idea principale è stata quella di andare ad aumentare il **rumore digitale** all'interno dell'immagine considerata: sostituire quindi il pixel attualmente valutato, con uno dei pixel vicini, all'interno di una determinata finestra spaziale predefinita. Un pixel è definito come rumore se e solo se è il massimo o il minimo all'interno proprio della suddetta finestra.
Il Noise è sostanzialmente l'effetto che si può ottenere utilizzando una fotocamera con sensori di bassa qualità. Nel comando che segue è mostrato come trasformare l'immagine applicando il rumore massimo. Esistono diverse varianti di rumori applicabili quali il Gaussiano, di Impulso, Laplaciano, Moltiplicativo e di Poisson.

```
magick test_image/image8.jpg -statistic NonPeak 100
   test_image_100/image8proc.jpg
```



(a) *Immagine Originale*

(b) *Immagine con Noise σ=15-PSNR=19*

(c) *Immagine con Noise σ=40- PSNR=16*

(d) *Immagine con Noise σ=100- PSNR=14*

Figura 6.4: Immagine con applicazione della riduzione del Noise

Nella figura 6.4 è chiara l'evoluzione di una immagine che subisce l'applicazione di un Noise crescente. Qui il peggioramento sarà molto più visibile all'occhio umano. La terza immagine c) corrisponde al punto di rottura: da qui la rete predirrà con percentuali via via scarsissime fornendo dapprima "cuffia da piscina" e poi "peperone rosso".

Variazione di Contrasto

La variazione di Contrasto non può essere definita come una distorsione vera e propria ma solo come una modifica/variazione delle condizioni di una immagine; infatti viene aumentato o ridotto il Contrasto di una immagine lavorando sulla differenza di intensità tra i soggetti più chiari o più scuri. Il tool ImageMagick va a lavorare simultaneamente su luminosità e Contrasto, variando con lo stesso comando l'uno e l'altra. Chiaramente, settando a zero o non settando la luminosità si può fare variare solamente il Contrasto. Questo è quello che è stato fatto nel nostro caso.

```
magick test_images/image10.jpg -brightness-contrast -0,20
   test_image_20/image10proc.jpg
```

Le immagini saranno parecchio modificate fin dai primi step di applicazione delle modifiche, ma manterranno a lungo la definizione dei soggetti rappresentati. Infatti, ad esempio nella figura 6.5 il punto di rottura sarà identificabile a σ=90, quindi quasi al massimo della "distorsione".

(a) *Immagine Originale*

(b) *Immagine con variazione di Contrasto σ=50- PSNR=14*

(c) *Immagine al punto di rottura conσ=90- PSNR=11*

(d) *Immagine con Contrasto massimo σ=100- PSNR=10*

Figura 6.5: Immagine a cui è stata applicata una variazione del Contrasto

6.4 Analisi dei dati ottenuti: Classificazione

L'immagine raffigurante un giocatore di baseball rappresentata in 6.5 può rientrare in una delle casistiche da analizzare dal momento che si tratta di una immagine con soggetto principale ben definito e per la quale la prima predizione di classificazione mostra delle percentuali molto alte, anche modificandone la qualità: il punto di rottura si assesta in situazioni di peggioramento elevate quindi necessitano aumenti consistenti di livelli di distorsione, come è visibile in figura 6.6 dove sono mostrati i punti di rottura per ciascuna delle "distorsioni testate" per la suddetta immagine. Nello specifico le curve rappresentano l'andamento delle percentuali di predizione.
Il punto di rottura per il Blur si assesta intorno ad un PSNR pari a 22.35, nel punto in cui la curva della prima predizione si interseca con la curva della seconda predizione; un comportamento molto simile, escludendo un leggero picco è presente con il Noise dove il punto di rottura si assesta al PSNR pari a 20; per Contrasto e riduzione della qualità invece si notano delle curve più o meno costanti. Per quanto riguarda il punto di rottura, dall'analisi dei dati in possesso, si evince come per il Contrasto non sia rilevato mentre per la riduzione del JPEG si assesti ad un PSNR pari a 10, quindi a condizioni ben lontane dalle originali. Quindi, già ad occhio in prima analisi, pur avendo analizzato un caso semplificato, si può provare ad affermare che la rete soffre maggiormente il Blur ed il Noise, piuttosto che le altre due distorsioni.

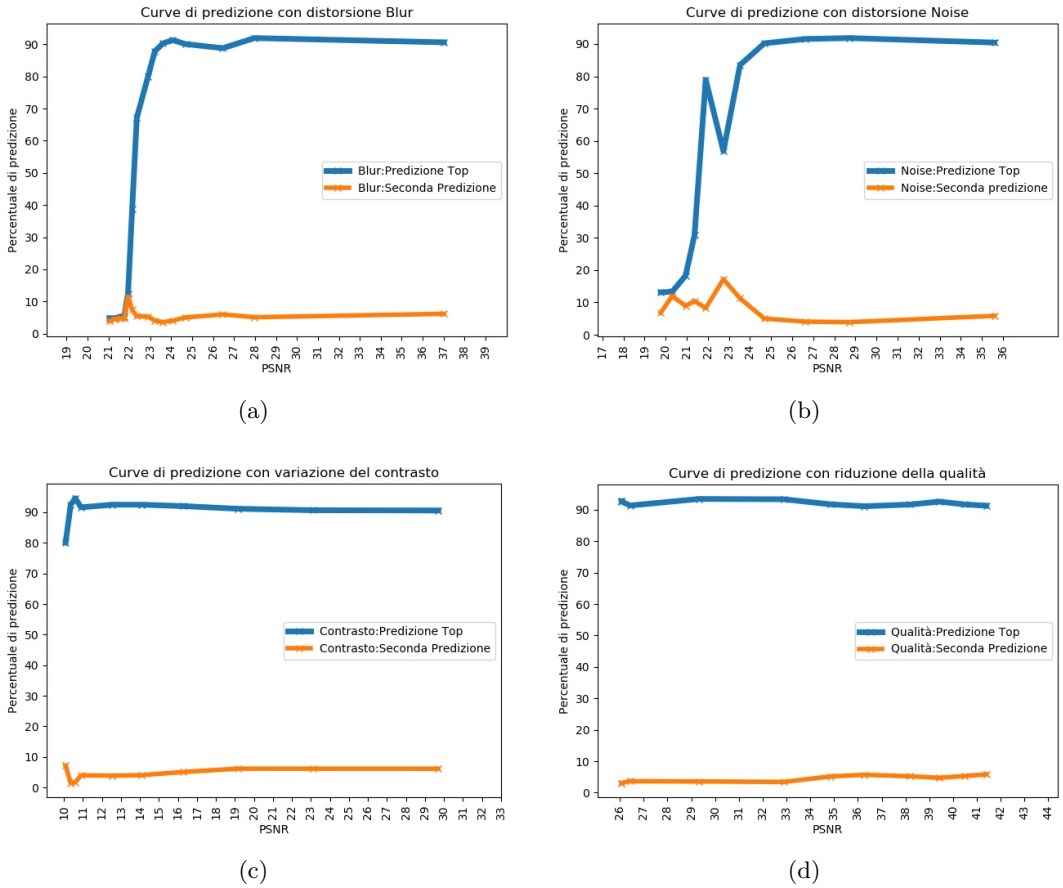


Figura 6.6: Curve di predizione per l'immagine 6.4

La stessa analisi, per provare a dare una riprova di quanto appena detto, può essere realizzata su un'immagine di base totalmente diversa e con sfaccettature diverse anche in termini pratici: ad esempio una immagine che può ammettere label generiche e label più specifiche.
L'immagine in figura 6.7 produce una predizione primaria (in condizioni originali) del 68% che indica "gatto tigrato" e come seconda predizione indica "gatto soriano" per il 23%. I dati ottenuti nei relativi grafici potrebbero essere fuorvianti ed imprevedibili dal momento che le due label sono interscambiabili e quasi equivalenti. In questo caso il comportamento della rete è influenzato dalle definizioni delle classi presente nel file di label usato. Probabilmente usando un file di label ridotto, con categorie meno specifiche avremmo una prima predizione parecchio elevata equivalente alla somma delle due sopra citate, con percentuale totale circa del 91%. Questo ci fa immaginare un andamento della curva similare all'immagine 6.4.

Figura 6.7: Immagine in qualità originale con label doppia

Analizzando l'andamento delle curve in figura 6.8 si nota come l'oscillazione tenda a farle apparire come curve errate anche se in realtà, alla luce di quanto detto in merito alle label, immagini di questo tipo costituiscono un "falso positivo" per la nostra fase di analisi dei dati. Falso positivo proprio perchè si ha l'impressione che sia stato inviduato un punto di rottura nonostante non lo sia realmente.
Il punto di rottura con l'applicazione del blurring è fissato al 51%, che come si osserva dal grafico, è il punto in cui si crea un picco sulla seconda curva: qui le prime due label si invertiranno, una diventerà la prima predizione e l'altra la seconda. Di base in questo caso quindi le predizioni saranno mediamente più basse rispetto alla media delle predizioni ottenute con una rete così performante. Il Noise tendenzialmente è meno oscillante rispetto al Blur, ma il comportamento ottenuto è similare visto che già alla prima applicazione della distorsione è possibile incontrare il punto di rottura: esso staziona intorno al PSNR=23, con percentuale di predizione pari al 45%. Contrasto e riduzione della qualità, di contro, presentano delle curve più morbide ma sempre anomale rispetto agli standard ottenuti con queste due "distorsioni" su altre immagini: tutto ciò, sottolinea ancora una volta come entrambe di per sè abbiano meno impatto sul riconoscimento ma conferma anche che la scelta del file label sia un elemento di grande impatto. Il Contrasto, infine, presenta un punto di rottura chiaramente identificabile sul grafico nel punto in cui le due curve si toccano mentre invece per la qualità staziona ad un PSNR=21, punto in cui vi è la massima applicazione di distorsione. Di seguito in figura 6.7 sono rappresentati i grafici di cui è stata fatta adesso una breve panoramica, grafici chiaramente poco validi ma utili al riconoscimento di un nuovo pattern comportamentale.

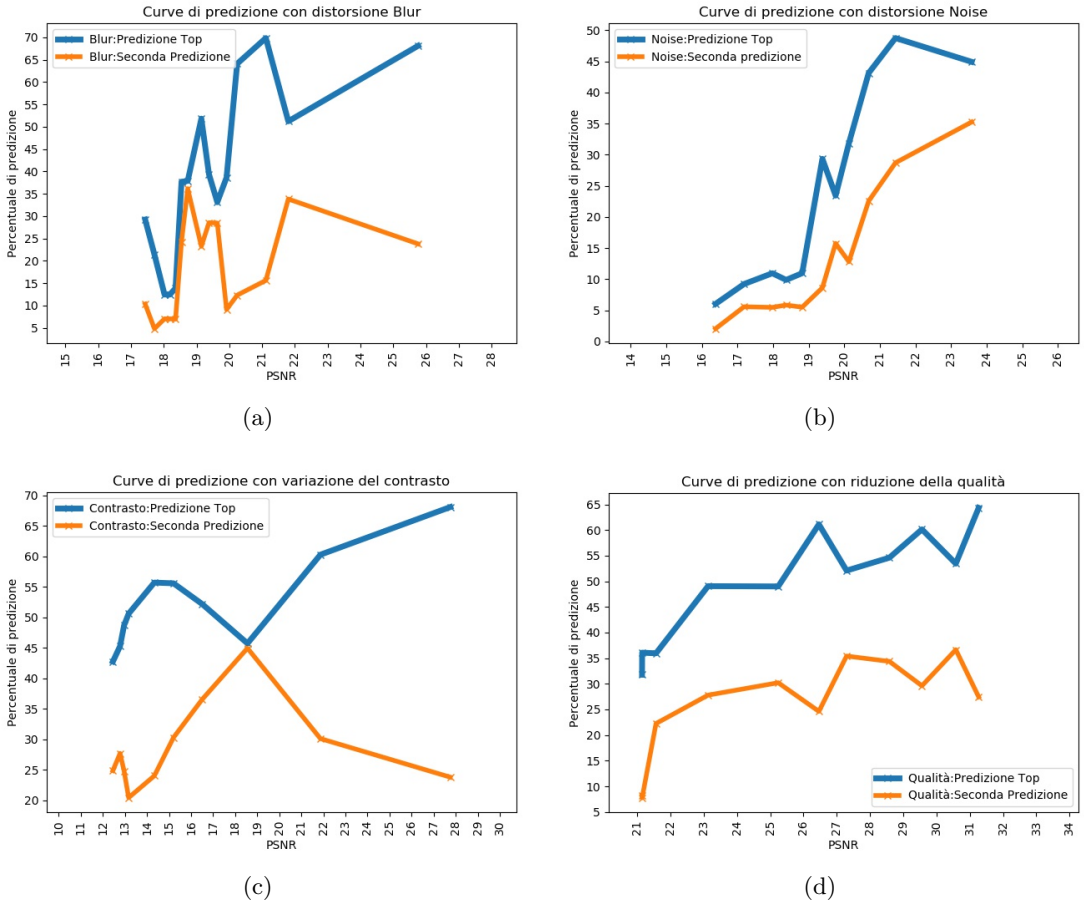


Figura 6.8

Per avvalorare quanto detto finora, è stato necessario lavorare su una quantità più ampia di immagini in modo da ottenere una stima quantitativa oltre che qualitativa.
Sono stati calcolati i punti di rottura di 250 immagini per ciascuna delle 4 distorsioni, tracciando dei grafici di dispersione, visibili in figura (6.9).
Su 250 immagini infatti 247 immagini presentano punti di rottura per il Blur, 249 per il Noise, 195 per la riduzione di qualità e 180 per variazione di Contrasto. Quindi la rete presenta più punti di rottura come prevedibile per Blur e Noise e meno per qualità e Contrasto.
Osservando invece i dati numerici mostrati sempre in figura 6.9, per Blur e Noise si nota una maggiore densità intorno ai valori di PSNR che oscillano tra 22 e 24 e che si trovano buone densità anche su valori molto alti di PSNR: questo dimostra proprio che vi sono dei punti di rottura anche già a PSNR alti, quindi su immagini che sono molto somiglianti alle originali. La riduzione di qualità invece ha dei punti di rottura localizzati su intervalli di PSNR ristretti, con picchi di densità intorno a PSNR=24 mentre la riduzione del Contrasto mostra punti di rottura chiaramente localizzati a PSNR decisamente bassi.

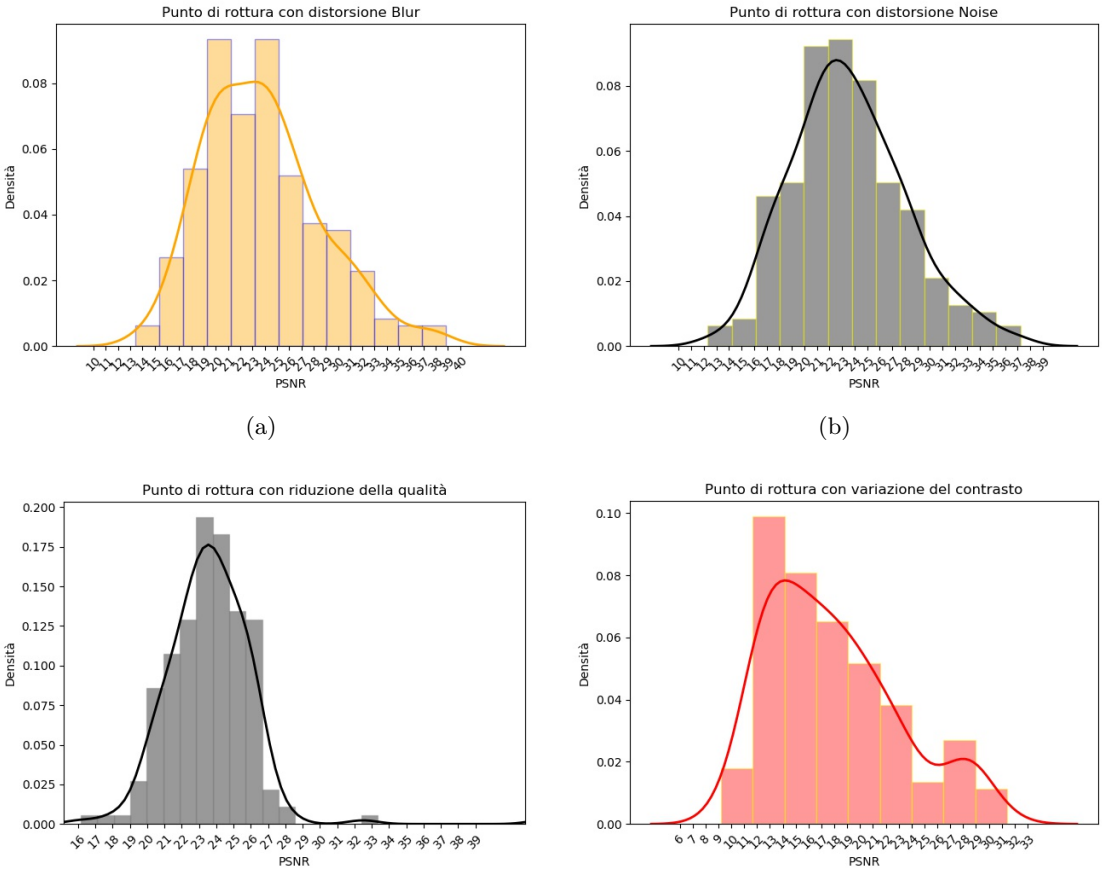


Figura 6.9

Per rendere ancora più tangibili i dati esposti, sono stati "plottati" dei grafici che mostrano le frequenze dei punti di rottura per ogni distorsione: questi grafici 6.10 6.11 sono molto simili ai grafici di dispersione 6.9, ma offrono un'idea graficamente più chiara dei pattern comportamentali della rete.

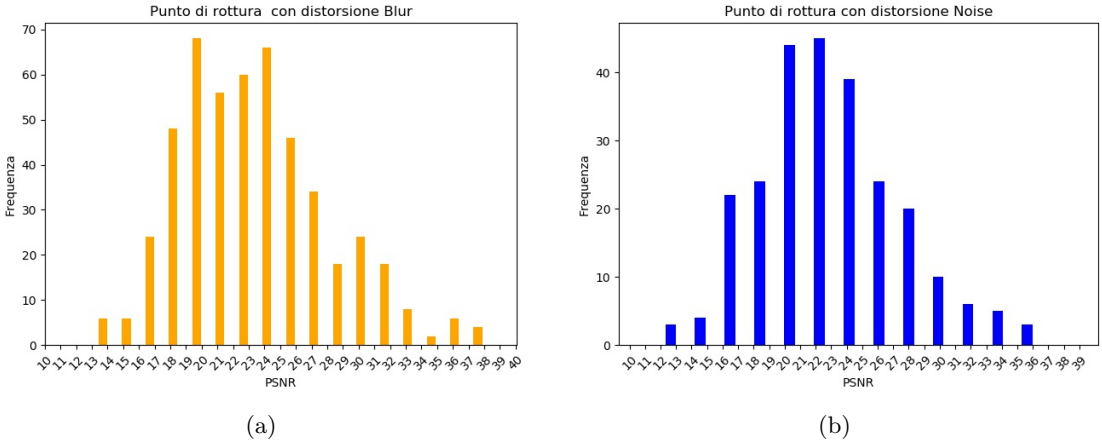
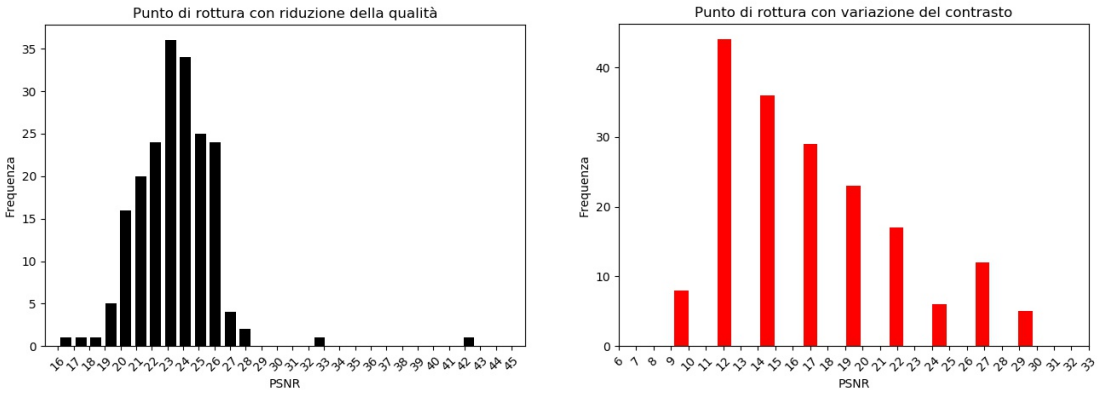


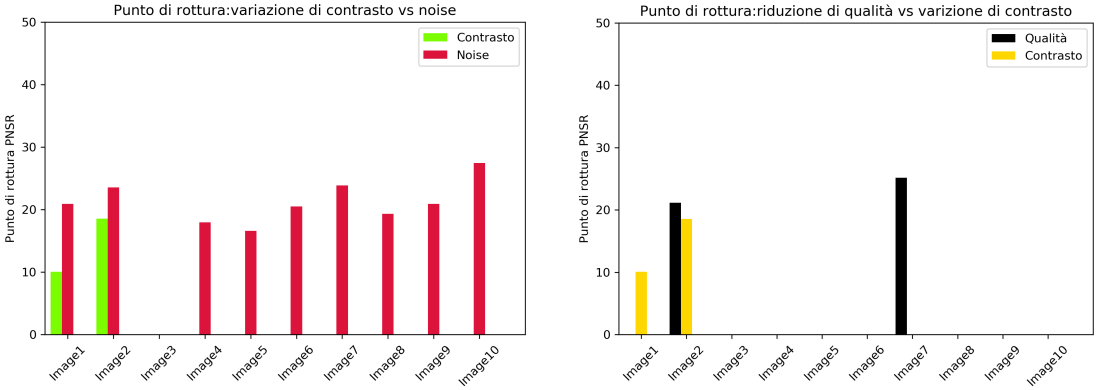
Figura 6.10: I grafici di frequenza che sottolineano quanto detto finora per blurring e Noise.



(a) (b)

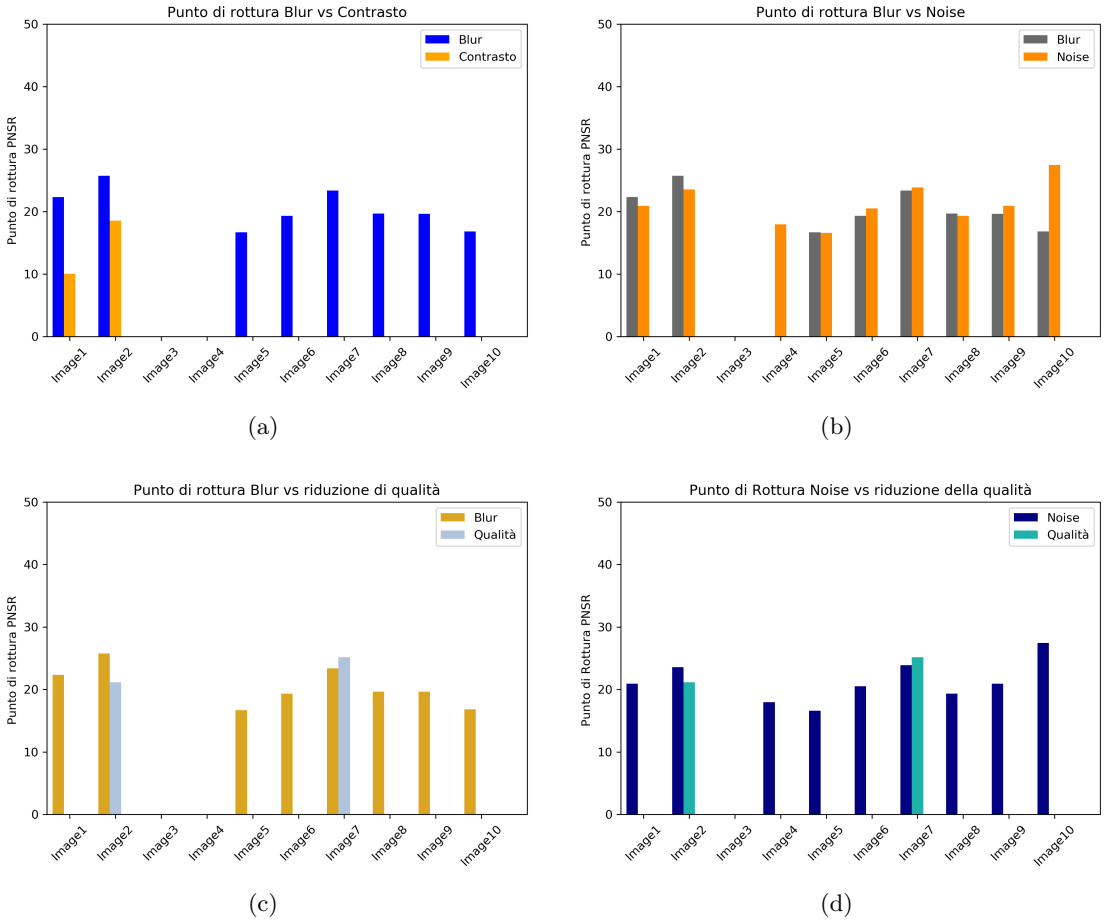
Figura 6.11: I grafici di frequenza che sottolineano quanto detto finora per riduzione di qualità e variazione del Contrasto

Successivamente ci si è focalizzati ancora su un altro tipo di analisi: un confronto tra le varie distorsioni in merito ai punti di rottura tarato su 10 immagini accuratamente selezionate, cosa che pur apparendo quantitativamente poco intuitiva fornisce comunque un valido termine di paragone.



(a) (b)

Figura 6.12: Prima parte dei grafici di confronto tra le diverse distorsioni, tarati su 10 immagini.



(a) (b)

(c) (d)

Figura 6.13: Seconda parte dei grafici di confronto tra le diverse distorsioni, tarati su 10 immagini.

È visibile, come vi sia un netto divario tra Blur contro Contrasto, così come in Blur contro Qualità e Noise contro qualità e come vi sia un grosso equilibrio per Blur contro Noise. In Contrasto contro Noise, nello specifico, si nota un grosso dislivello a favore di Noise mentre il grafico Qualità contro Contrasto sottolinea ancora una volta che entrambe sono le distorsioni meno affette.

6.5 Analisi dei dati ottenuti: Object Detection

La fase di testing sul task di Object Detection è stata impostata sulla falsa riga di quanto fatto nel primo task, ovviamente adattandosi al contesto ben diverso. Per quanto riguarda i dettagli tecnici sulla rete scelta e sulla tecnica Faster R-CNN utilizzata, la lettura è rimandata alle pagine in cui vengono esposti i dettagli tecnici, mentre qui ci si focalizzerà subito su quanto è variato rispetto all'altra fase di testing: ci saranno dei dati aggiuntivi e dei dati di base diversi dal momento che la rete in output fornisce la migliore predizione per ogni istanza di oggetto individuata oltre che il numero totale di oggetti localizzati per ogni immagine ed una immagine annotata con i box relativi agli oggetti/soggetti individuati. Prima di mostrare un primo esempio di localizzazione, come fatto per il primo task bisogna fare delle precisazioni per lo più tecniche. Il numero massimo di oggetti riconoscibili è stato settato a 15, in modo da evitare che venissero identificati tanti oggetti magari poco interessanti: alla luce di ciò è stato necessario inserire un nuovo tipo di pruning relativamente alla percentuale di predizione. Per cui è stato deciso di considerare come validi i box la cui prima predizione avesse una percentuale superiore al 60%. Chiaramente questi valori non sono stati scelti casualmente ma sono frutto di diversi test, durante i quali tali valori sono stati variati più volte al fine di trovare un buon compromesso sulla mole di dati ottenuti e sulla tempistica di elaborazione.
L'analisi dei dati inoltre è partita su un set di 10 immagini accuratamente selezionate con un criterio fondamentale: i soggetti/oggetti molteplici delle immagini dovevano essere chiaramente identificabili con dei casi in cui lo stesso oggetto fosse ripetuto e magari sovrapposto ad altri in modo da evidenziare su piccola scala come si comportasse la rete.

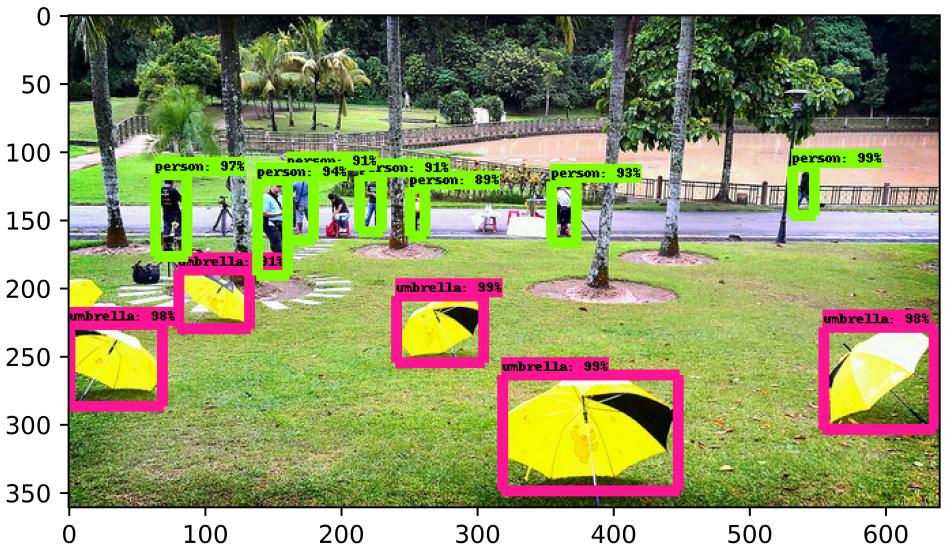


Figura 6.14: Esempio di immagine annotata in condizioni originali.

Elenco delle annotazioni che la rete ha fornito come output testuale sull'immagine in figura 6.14:

```
99.81% umbrella
99.19% umbrella
99.09% person
98.80% umbrella
98.40% umbrella
97.35% person
94.71% person
93.56% person
91.71% person
91.56% umbrella
```

Il dato che balza all'occhio è che la rete ha mostrato in output delle percentuali di predizione mediamente elevate, quasi tendenti alla perfezione, avvicinandosi spesso a valori tra 95% e il 99%. Diminuendo invece in maniera sostanziale la qualità visiva dell'immagine, il numero di oggetti localizzati è sceso fino a toccare lo zero: con predizioni sotto il 65% la rete ha fornito label poco veritiere, ha perso degli oggetti o ha proposto dei riquadri totalmente errati. In questa fase si proveranno ad abbozzare delle analisi qualitative tracciando delle curve che mostrano il numero di oggetti localizzati sulle ordinate e contestualmente il valore del PSNR sulle ascisse. Le pendenze delle curve hanno proposto dei dati parecchio interessanti da cui ricavare ancora dei veri e propri pattern comportamentali.
Le "distorsioni" utilizzate sono le medesime rispetto a prima cioè Blur, Noise, variazione di Contrasto e riduzione della qualità JPEG.

Figura 6.15: Immagine originale, gli oggetti individuati sono 6, 3 bottiglie, 2 banane e un tavolo da cucina.

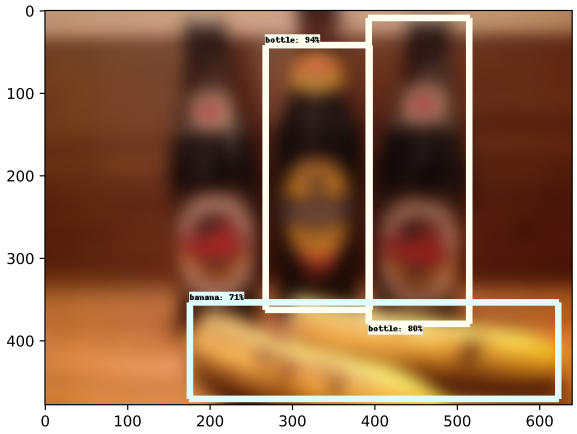


Figura 6.16: Immagine con σ=10 e PSNR=19.53: gli elementi localizzati ora sono 3.

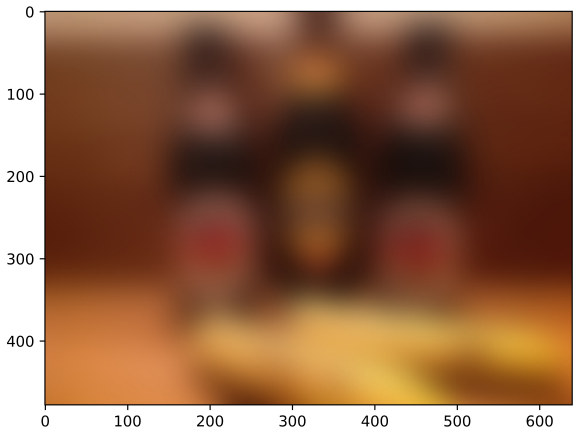


Figura 6.17: Immagine con σ=20 e PSNR=17: gli elementi localizzati alla qualità minima sono zero.

L'analisi visiva delle immagini presentate dalla figura 6.15 alla 6.17 mostra una base comportamentale del blurring analizzabile subito grazie ai grafici mostrati di seguito in figura 6.18. Il Blur e il Noise mostrano una curva simile: scende in maniera lenta e progressiva, quindi c'è un peggioramento dell'operatività della rete. Il tutto chiaramente va correlato alle percentuali di predizione per ogni singolo box. Infatti, l'operazione in oggetto non è altro che lo step successivo rispetto alla classificazione e di conseguenza le due operazioni vanno analizzate in maniera parallela: l'andamento delle percentuali sarà concorde a quanto detto nella prima fase di analisi dei dati per cui al dato del numero di elementi evidenziati va correlata anche la precisione con cui quel dato box è classificato. Questa breve affermazione tornerà molto utile.
La rete per Blur e Noise evidenzierà un numero di oggetti localizzati minimo ad un PSNR che oscilla intorno ai valori 18 e 19 quindi immagini lontane dalla qualità originale massima ma non troppo, dal momento che il PSNR in cui si evidenzia il numero massimo di oggetti oscilla tra 26 e 27 per entrambe le distorsioni.
Per quanto riguarda invece variazione di qualità e Contrasto, è visibile che si tratti di

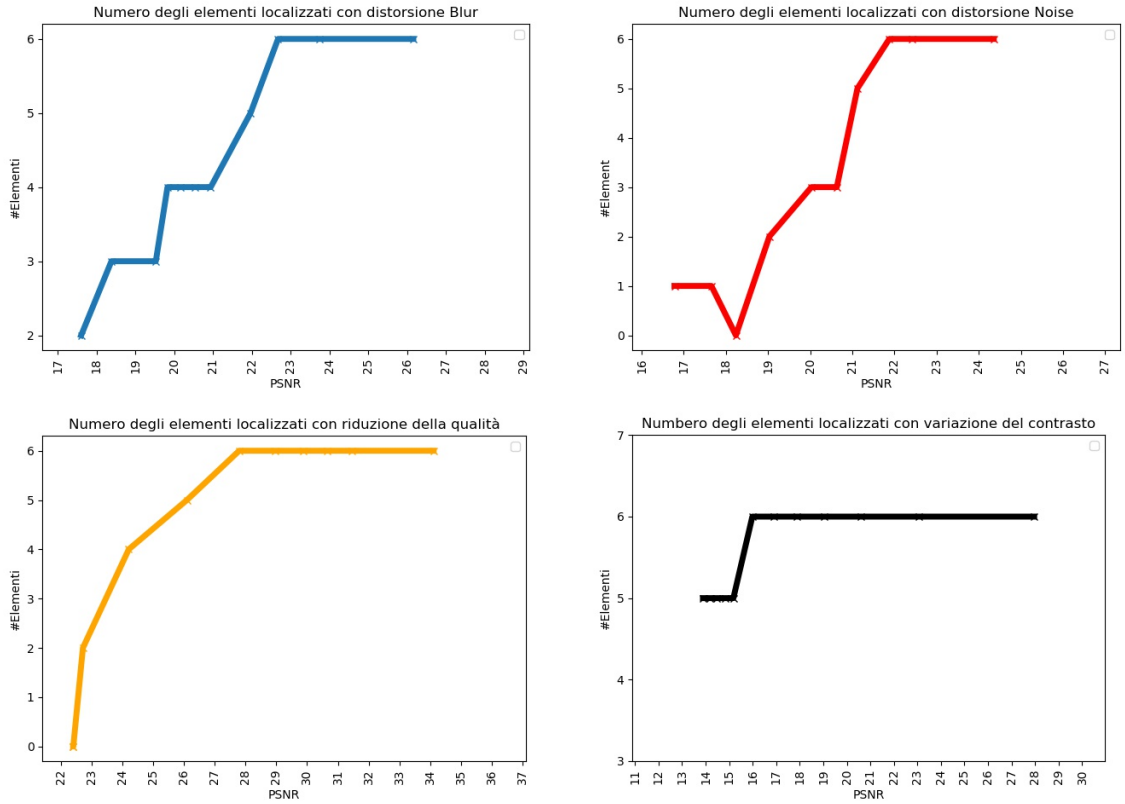


Figura 6.18: Questi grafici mostrano con delle curve l'andamento delle localizzazioni al variare del PSNR.

curve che scendono in maniera più morbida. Il Contrasto mantiene un numero costante ed alto di elementi localizzati anche con PSNR veramente bassissimi; per la variazione di qualità JPEG il contatore di oggetti localizzati sarà pari a zero ad un valore di PSNR=22 circa.
In una prima analisi parrebbe che i pattern comportamentali rispecchino quelli ottenuti per la classificazione, con il Contrasto che appare anche qui quello che fa soffrire meno in termini di operatività la rete ma con il solo cambiamento della variazione di qualità che sembra avvicinarsi al comportamento di Blur e Noise. Servirà evidenziare altri casi per avere la conferma di tutto ciò.
Prima di analizzare un'altra immagine, totalmente differente, bisogna sottolineare come nel grafico del Noise, sia apparso un falso positivo: questo va a confermare positivamente la scelta descritta in merito al taglio delle localizzazioni, che mostrassero predizioni basse al diminuire della qualità visiva delle immagini. Dopo aver raggiunto zero localizzazioni ad un PSNR pari a 18 sul grafico si nota un picco di salita ad un PSNR inferiore: è chiaro che è stato localizzato erroneamente un oggetto, come nell'esempio in figura 6.19.

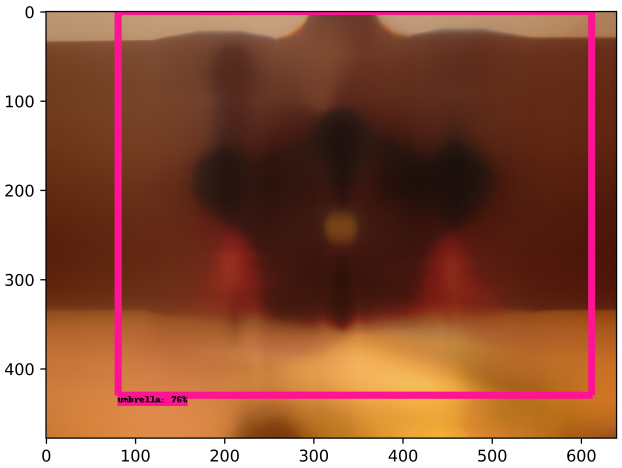


Figura 6.19: Immagine di un falso positivo, con un oggetto erroneamente localizzato come umbrella. Il PSNR è pari a 18.

Un altro esempio di immagine è visibile in figura 6.20, utile ad avere conferma dell'analisi precedentemente ottenuta.

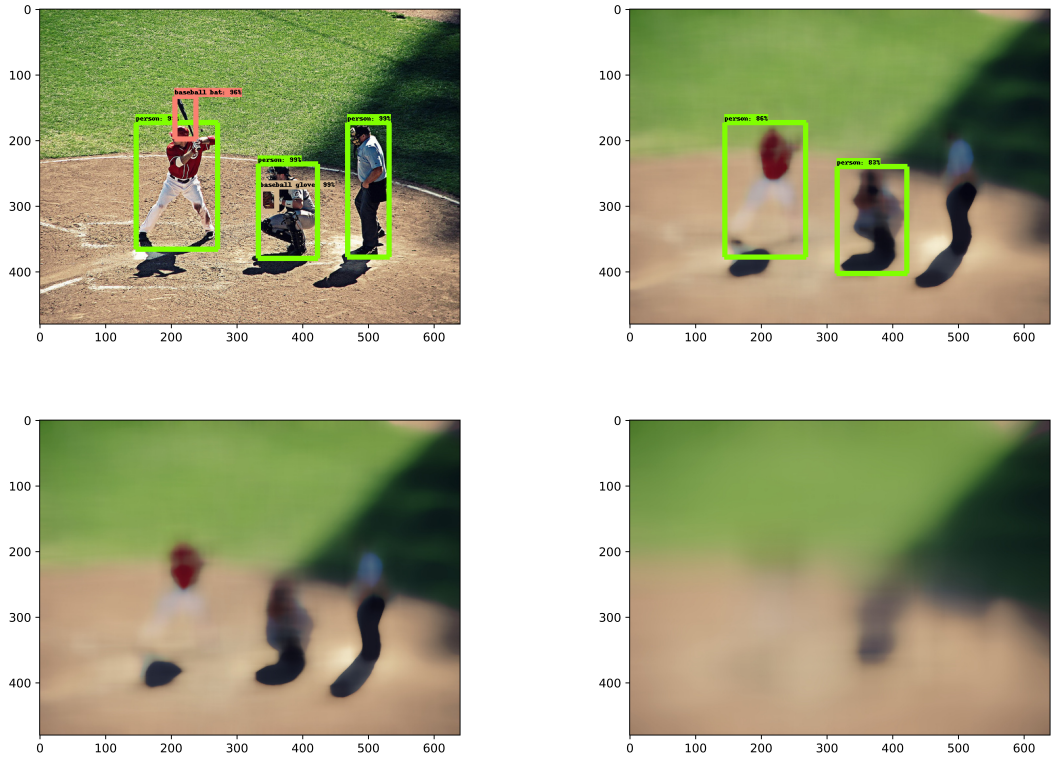


Figura 6.20: Secondo esempio di immagine: qui con l'applicazione del Noise le prestazioni peggiorano sensibilmente. Nella prima in alto a sinistra ho l'immagine originale, poi a destra ho PSNR=17, in basso a sinistra PSNR=16 e destra PSNR=15.

I grafici in figura 6.21 per l'immagine in figura 6.20, pur mostrando un andamento altalenante, che possibilmente nasce dalla natura dell'immagine, evidenziano quanto visto anche con l'altra immagine: Blur e Noise vanno ancora di pari passo, dando sempre un disturbo quasi sempre similare alla rete mentre il Contrasto appare distaccarsi anche qui nettamente rispetto ai due precedenti citati, permettendo alla rete, anche con ampie applicazioni di distorsione una discreta localizzazione. Viene confermato che la riduzione di qualità JPEG, che nella classificazione, andava a braccetto con il Contrasto qui si discosta da quest'ultima, avvicinandosi invece al comportamento di Blur e Noise. Tale pattern comportamentale sarà visibile in tutti i test fatti su svariate immagini.

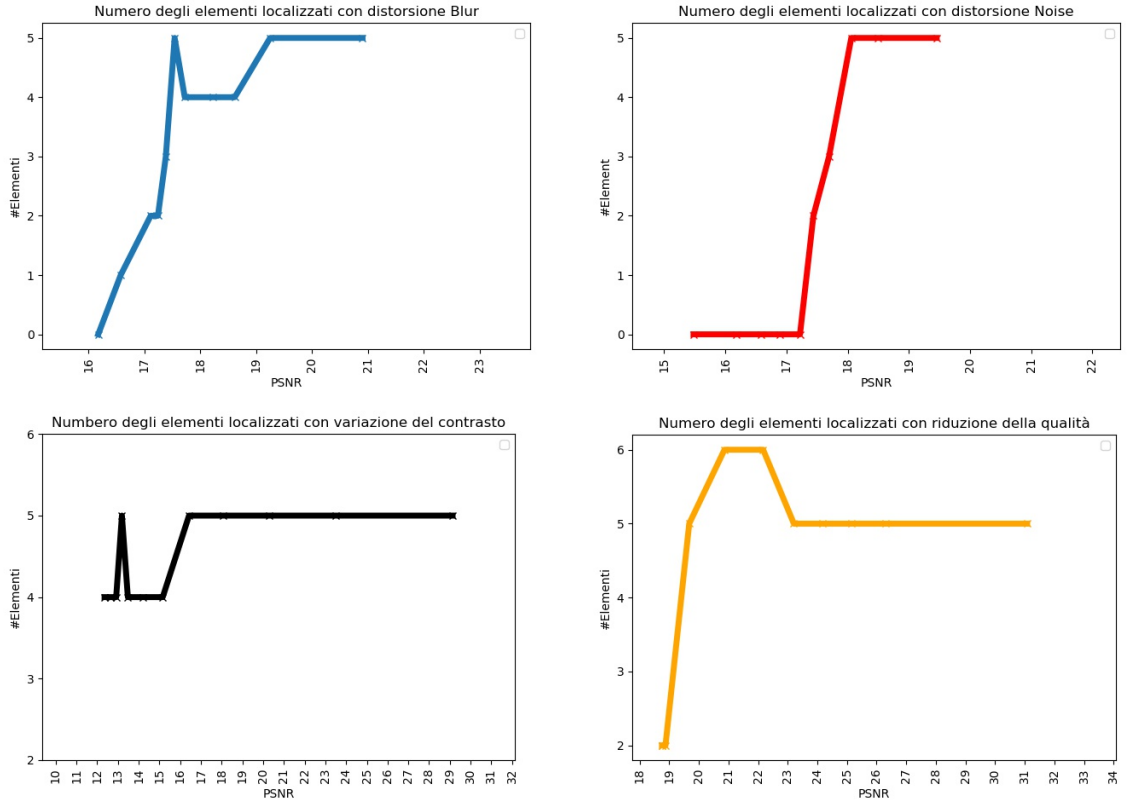


Figura 6.21: Questi grafici mostrano con delle curve l'andamento delle localizzazioni al variare del PSNR per l'immagine in figura 6.20.

Finora è stata fatta una analisi, analizzando il campione di una singola immagine per volta e nonostante ciò è stato evidenziato un pattern comportamentale ben delineato. Per dare però una riprova si è cercato di utilizzare un grafico che mostrasse la dispersione dei valori di localizzazione per un determinato set di immagini lavorando su set di 5, 10, 50 immagini. Saranno mostrati per semplicità di seguito solo i grafici inerenti ad un batch di 5 immagini.

I suddetti grafici mostrati nelle figure 6.22, 6.23, 6.24, 6.25 confermano quanto affermato in precedenza. Di seguito i grafici per Blur e Noise.

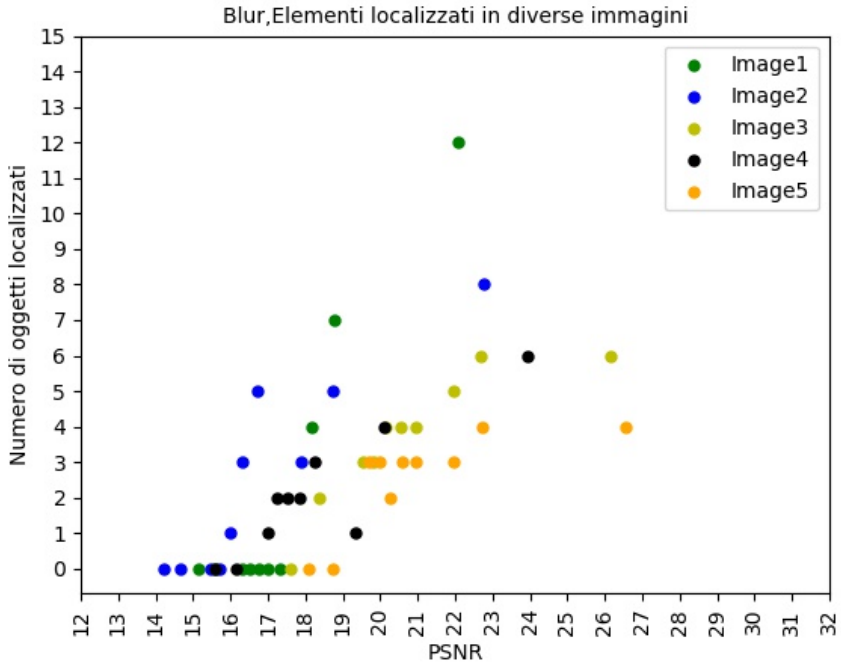


Figura 6.22: Scatter Plot per il Blur.

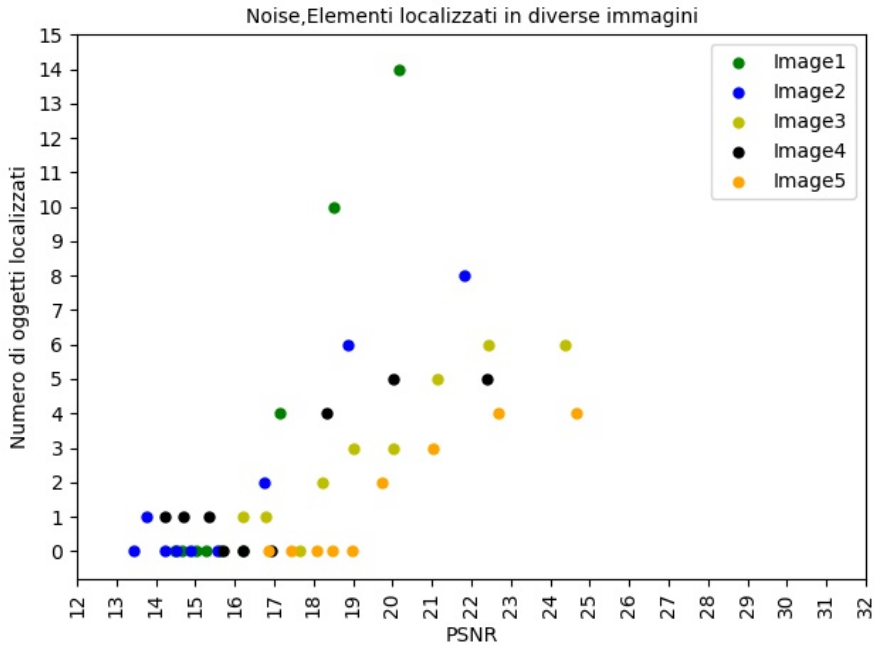


Figura 6.23: Scatter Plot per il Noise.

In questa pagina invece sono evidenziati i grafici per riduzione di qualità e variazione di Contrasto.

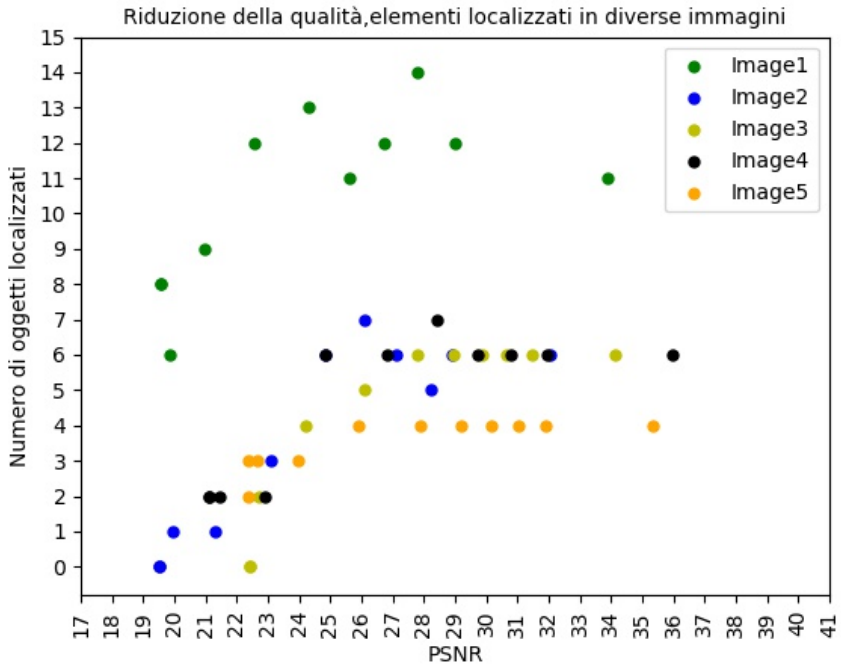


Figura 6.24: Scatter Plot per la riduzione di qualità JPEG.

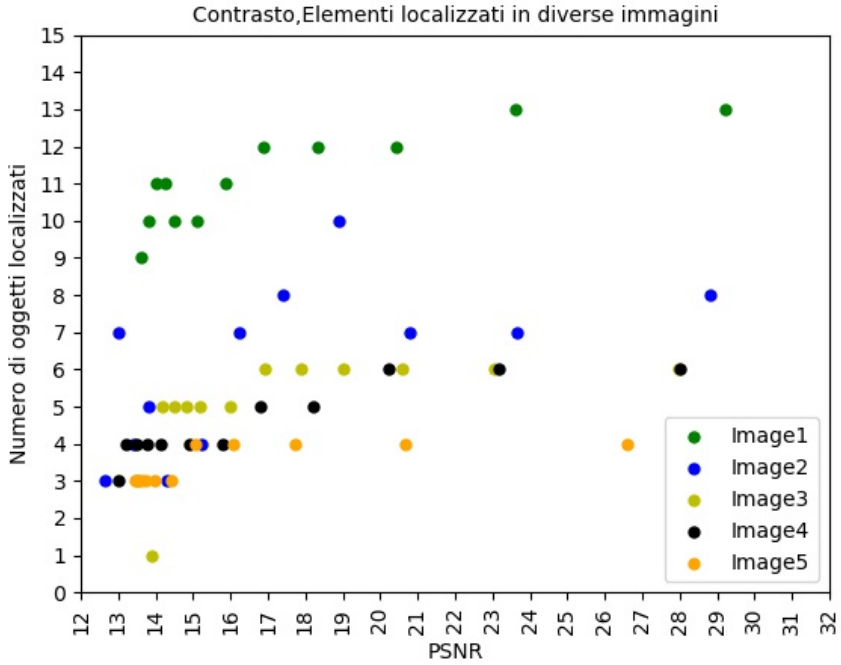


Figura 6.25: Scatter Plot per la variazione di Contrasto.

Capitolo 7

Conclusioni

Il percorso di questa tesi di ricerca, articolato in 6 capitoli più il seguente, ha messo in risalto le modalità di utilizzo delle reti neurali su immagini e video mostrando anche alcune possibili applicazioni che si possono compiere attraverso delle tecniche di intelligenza artificiale, che sempre più al giorno d'oggi stanno diventando di utilizzo comune. Così, a partire dai primi capitoli, si è cercato dapprima di fornire delle conoscenze basilari che dessero un buon background su quello che si sarebbe visto in termini applicativi, e poi si è cercato di fornire delle nozioni in merito allo stato dell'arte: Framework, Dataset, API, possibili applicazioni pratiche.
I task principali su cui si è incentrato lo studio applicativo sono stati classificazione degli oggetti, Object Detection e anche segmentazione semantica degli oggetti nelle immagini quindi in sintesi si è potuto parlare di **object recognition in images**.
Queste tecniche oggi sono utilizzabili per ricerche su siti di compra-vendita online, per i social network e per tante altre applicazioni che spesso sono davanti ai nostri occhi ma di cui magari non si conosce il background tecnologico, quindi è parso interessante approfondirle e comprenderne i meccanismi di funzionamento.
Ma lo scopo vero e proprio è stato testare tali tecniche per rendersi conto di quanto ci si stia avvicinando sempre più al comportamento dell'essere umano e nello specifico del cervello umano. Nella trattazione, proprio per fare ciò, è stato portato avanti uno studio di analisi e testing: questo è stato proprio il nucleo centrale della nostra ricerca.
Come si comporta una rete se si varia l'input?
Come si comporta la rete se si varia la qualità dell'input?

Già nella fase sviluppo si era notato come con l'evoluzione, reti più o meno moderne o più o meno diverse tra loro, proponessero comportamenti più o meno accurati in termini di risultati: allora partendo da queste osservazioni, dalle domande precedentemente poste e dal background acquisito nel corso dello studio di ricerca iniziale si è iniziato a testare una rete, scelta appositamente, con immagini in qualità originale contenenti soggetti singoli, prima ben definiti e poi molteplici e magari confusionari. Il passo successivo e finale è stato quello di lavorare con immagini in qualità inferiore rispetto a quella originale, applicandovi degli effetti che creassero dei peggioramenti visivi. Le distorsioni selezionate cioè Blur, Noise, variazione del Contrasto, riduzione della qualità JPEG, applicate su batch di 10, 50, 250 immagini (a seconda del task) si sono rivelate un buon compromesso per il nostro scopo fornendo dei risultati chiari e tangibili.
Tra le prime osservazioni bisogna sottolineare come la rete scelta, la Inception ResNet v2, reagisce in maniera differente alle diverse distorsioni ma anche al variare del task: in particolare il Blur e il Noise impattano maggiormente la rete che già con leggere distorsioni inizierà a produrre errori e falsi risultati; variazione del Contrasto e riduzione della qualità

JPEG invece, seppur disturbando la rete sono meno impattanti.
Qui però bisogna fare una leggera distinzione, osservata durante il testing: il Contrasto sia per la classificazione e la localizzazione mantiene un comportamento lineare mentre la riduzione della qualità JPEG, che per la classificazione seguiva un comportamento simile al Contrasto, invece per la localizzazione risulta essere di maggiore impatto avvicinandosi al Noise ed al Blur.
Ottenuti questi risultati, si è notato quindi che pur tendendo ad avvicinarsi all'osservazione dell'occhio umano, le reti hanno ancora delle pecche di precisione, ma visti i progressi fatti nell'ultimo decennio l'auspicio è quello che un giorno possano essere sempre più precise in modo tale da non soffrire in maniera così impattante determinati peggioramenti visivi degli input. Il percorso quindi, ha confermato quanti passi in avanti sono stati fatti, visto che si tratta di un contesto in continua evoluzione che a partire dal 2012 ha subito una grossa accelerazione in termini di "improvement'. I task approfonditi in questa tesi, pur essendo di dominio comune negli ambienti scientifici non sono stati studiati in maniera approfondita in merito all'analisi comportamentale delle reti al variare delle distorsioni e della loro applicazione: è qualcosa che sta iniziando a prendere piede solo nell'ultimo biennio. Sarebbe quindi interessante iniziare magari a delineare degli standard comportamentali più ampi, utilizzando svariate distorsioni diverse tra loro; ma si potrebbero iniziare ad approfondire task meno noti o più complicati in termini di sviluppo, come la segmentazione o l'analisi dei soggetti/oggetti all'interno dei video, per poi intraprendere degli studi simili a quello sviluppato in questa trattazione.

Bibliografia

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

[2] Y. LeCun, Y. Bengio, G. Hinton. (2015). *"Deep learning"*. Nature, vol. 521, no. 7553, pp. 436–444.

[3] "https://vitolavecchia.altervista.org/caratteristiche-e-differenza-tra-machine-learning-e-deep-learning/"

[4] "https://www.fokus.fraunhofer.de/en/fame/workingareas/ai"

[5] Ian Goodfellow and Yoshua Bengio and Aaron Courville. *"Deep learning"*. MIT Press, 2016. "http://www.deeplearningbook.org"

[6] "https://it.wikipedia.org/wiki/"

[7] "https://medium.com/topic/machine-learning"
"http://www.academia.edu/Documents/in/Machine_Learning"
"https://towardsdatascience.com/"

[8] Hinton, G.E. *"How neural networks learn from experience"*. Scientific American, September 1992, pp. 145–151

[9] "https://italiancoders.it/deep-learning-svelato-ecco-come-funzionano-le-reti-neurali-artificiali/"

[10] "https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f"

[11] "http://www.ce.unipr.it/people/medici/geometry/node107.html"

[12] "https://stepupanalytics.com/detailed-introduction-to-recurrent-neural-networks/"

[13] "https://towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2"

[14] "https://seas.ucla.edu/ kao/nndl/lectures/cnn.pdf"

[15] Kaplanoglou, Pantelis. (2017). *"Content-Based Image Retrieval using Deep Learning"*. 10.13140/RG.2.2.29510.16967.

[16] K. He, X. Zhang, S. Ren, and J. Sun. (2014). *"Spatial pyramid pooling in deep convolutional networks for visual recognition"*. In ECCV.

[17] Di Lella Michele Francesco. *"Stima della profondità da singola immagine per mezzo di una CNN addestrata mediante tecniche di computer graphics"*. Alma Mater Studiorum - Università di Bologna.

[18] "https://www.spaghettiml.com/2017/09/27/gradient-descent-o-discesa-del-gradiente/"

[19] Giacomini Elia (2016). *"Convolutional Neural Networks per il riconoscimento di nudità."*

[20] Ioffe, Sergey, and Christian Szegedy. (2015). *"Batch normalization: Accelerating deep network training by reducing internal covariate shift"*. In ICML.

[21] Srivastava, Nitish, Hinton, Geoffrey, Krizhevsky, Alex,Sutskever, Ilya, and Salakhutdinov, Ruslan. *"Dropout: A simple way to prevent neural networks from overfitting.* J. Mach. Learn. Res., 15(1):1929–1958, January 2014.

[22] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. (2014) *"Imagenet large scale visual recognition challenge"*. arXiv:1409.0575

[23] A. Berg, J. Deng, and L. Fei-Fei. *"Large scale visual recognition challenge 2010"*. "http://image-net.org/challenges/LSVRC/2010/pascal_ilsvrc.pdf"

[24] Lowe, D. G. (2004). *"Distinctive image features from scale-invariant keypoint"*. IJCV, 60(2): 91–110.

[25] Li, Ran & Li, Xuezhen & Kurita, Takio. (2015). *"Soft local binary patterns"*. 70-75. 10.1109/SOCPAR.2015.7492786.

[26] T. Ojala, M. Pietikainen and D. Harwood. (1996). *"A comparative study of texture measures with classication based on feature distributions"*. Pattern Recognition, vol. 29, no. 1, pp. 51-59.

[27] Cortes, Corinna & Vapnik, VN. (1995). *"Support Vector Networks"*. Machine Learning. 20. 273-297.

[28] Ahonen, Timo, and Matti Pietikinen, *"Soft histograms for local binary patterns"*. In Proceedings of the Finnish signal processing symposium, FINSIG, Vol. 5. 2007

[29] Perronnin F., Dance C. R. (2007). *"Fisher kernels on visual vocabularies for image categorization"*. In Computer Vision and Pattern Recognition (CVPR).

[30] Perronnin F., Sanchez J., Mensink T. (2010) *"Improving the Fisher kernel for large-scale image classification"*. In ECCV, 2010c

[31] Sanchez, J. and Perronnin, F. (2011). *"High-dim. signature compression for large-scale image classification"*. In CVPR.

[32] R.M. Gray and D.L. Neuhoff, *"Quantization"*. IEEE Trans. Inform. Theory, vol. 44, pp. 2325–2383, Oct. 1998.

[33] H. Jegou, M. Douze, and C. Schmid. (2011). *"Product quantization for nearest neighbor search"*. In TPAMI.

[34] Yashima Ahuja & Sumit Kumar Yadav. (2012). *"Multiclass Classification and Support Vector Machine"*. Global Journal of Computer Science and Technology Interdisciplinary. 12(11),14-20.

[35] van de Sande, K. E. A., Uijlings, J. R. R., Gevers, T., and Smeulders, A. W. M. (2011b). *"Segmentation as selective search for object recognition"*. In ICCV.

[36] van de Sande, K. E. A., Gevers, T., and Snoek, C. G. M. (2010) *"Evaluating color descriptors for object and scene recognition"*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 32(9): 1582-1596.

[37] "https://towardsdatascience.com/introduction-to-video-classification-6c6acbc57356"

[38] Lazebnik, S., Schmid, C., and Ponce, J. (2006). *"Beyond bags of features: Spatial Pyramid Matching"*.

[39] Subhransu Maji and Alexander C. Berg and Jitendra Malik, *"Classification Using Intersection Kernel Support Vector Machines is efficient"*. Proceedings of CVPR 2008, Anchorage, Alaska, June 2008.

[40] van de Sande, K. E. A., Gevers, T., and Snoek, C. G. M. (2011) *"Empowering visual categorization with the gpu"*. IEEE Transactions on Multimedia, 13(1):60-70,2011a

[41] Abien Fred Agarap. (2018). *"Deep learning using rectified linear units(relu)"*. In CoRR, abs/1803.08375.

[42] Shaeke Salman and Xiuwen Liu. (2019). *"Overfitting mechanism and avoidance in deep neural networks"*. InCoRR, abs/1901.06566.

[43] Zeiler, M. D. and Fergus, R. (2013). *"Visualizing and understanding convolutional networks"*. In CoRR, abs/1311.2901.

[44] Zeiler, M. D., Taylor, G. W., and Fergus, R. (2011). *"Adaptive deconvolutional networks for mid and high level feature learning"*. In ICCV.

[45] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2013). *"Overfeat: Integrated recognition, localization and detection using convolutional networks"*. In CoRR, abs/1312.6229.

[46] van de Sande, K. E. A., Snoek, C. G. M., and Smeulders, A. W. M. (2014). *"Fisher and vlad with flair"*. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

[47] Jasper R. R. Uijlings, Koen E. A. van de Sande, Theo Gevers e Arnold W. M. Smeulders (2013). *"Selective Search for Object Recognition"*. In International Journal of Computer Vision (IJCV).

[48] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. (2015). *"Going deeper with convolutions"*. In CVPR.

[49] Karen Simonyan e Andrew Zisserman (2014). *"Very Deep Convolutional Networks for Large-Scale Image Recognition"*. In CoRR

[50] He, Kaiming, et al. (2016). *"Deep residual learning for image recognition"*. Proceedings of the IEEE conference on computer vision and pattern recognition. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. (2016). *"Deep residual learning for image recognition"*. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 770-778

[51] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. (2015). *"Rethinking the Inception architecture for computer vision"*. arXiv preprint, 1512.00567, 2015. arxiv.org/abs/1512.00567.

[52] C. Szegedy, S. Ioffe, and V. Vanhoucke. *"Inception-v4, inception-resnet and the impact of residual connections on learning"*. In ICLR Workshop, 2016.

[53] R. Girshick. (2015). *"Fast R-CNN"*. In ICCV.

[54] S. Ren, K. He, R. Girshick, and J. Sun. (2015). *"Faster R-CNN: Towards real-time object detection with region proposal networks"*. In Neural Information Processing Systems (NIPS).

[55] "https://medium.com/@umerfarooq_26378/from-r-cnn-to-mask-r-cnn-d6367b196cfd"

[56] Hu, J., Shen, L. and Sun, G. (2018). *"Squeeze-and-excitation network"*. In CVPR

[57] "https://towardsdatascience.com/squeeze-and-excitation-networks-9ef5e71eacd7"

[58] Fukushima, Kunihiko. (1988). *"Neocognitron: A hierarchical neural network capable of visual pattern recognition"*. Neural networks 1.2, pp.119- 130

[59] LeCun, Yann, et al. (1998). *"Gradient-based learning applied to document recognition"*. Proceedings of the IEEE, 86.11, 2278-2324.

[60] "https://medium.com/@sh.tsang/paper-brief-review-of-lenet-1-lenet-4-lenet-5-boosted-lenet-4-image-classification-1f5f809dbf17"

[61] Alom, Md Zahangir, et al. (2018). *"The History Began from AlexNet: A vComprehensive Survey on Deep Learning Approaches"*. arXiv preprintarXiv:1803.01164.

[62] Krizhevsky, A., Sutskever, I., and Hinton, G. (2012). *"ImageNet classification with deep convolutional neural networks"*. In NIPS.

[63] Lin, Min, Qiang Chen, and Shuicheng Yan. (2013). *"Network in network"*. arXiv:1312.4400.

[64] Kataoka H, Iwata K and Satoh Y. (2015). *"Feature evaluation of deep convolutional neural networks for object recognition and detection"*. arXiv preprint arXiv:1509.07627.

[65] "https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202"

[66] "https://towardsdatascience.com/review-inception-v4-evolved-from-googlenet-merged-with-resnet-idea-image-classification-5e8c339d18bc"
[67] Zagoruyko, Sergey, and Nikos Komodakis. (2016). *"Wide Residual Networks"*. arXiv preprint arXiv:1605.07146
[68] "https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202"
[69] Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K. (2016). *"Aggregated residual transformations for deep neural networks"*. arXiv preprint arXiv:1611.05431.
[70] Veit, Andreas, Michael J. Wilber, and Serge Belongie. (2016). *"Residual networks behave like ensembles of relatively shallow networks"*. In Neural Information Processing Systems (NIPS).
[71] Abdi, Masoud, and Saeid Nahavandi. (2016). *"Multi-Residual Networks: Improving the Speed and Accuracy of Residual Networks"*. arXiv preprint arXiv:1609.05672.
[72] "https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4"
[73] X. Zhang, Z. Li, C. C. Loy, and D. Lin. (2017). *"Polynet: A pursuit of structural diversity in deep networks"*. In CVPR.
[74] Chollet F. (2016). *"Xception: Deep Learning with Depthwise Separable Convolutions"*. arXiv preprint arXiv:1610.02357v2.
[75] K.-H. Kim, S. Hong, B. Roh, Y. Cheon, and M. Park. (2016). *"PVANET: Deep but lightweight neural networks for real-time object detection"*. arXiv preprint arXiv:1608.08021
[76] G. Huang, Z. Liu, K. Q. Weinberger, and L. Maaten. (2017). *"Densely connected convolutional networks"*. In CVPR.
[77] "https://medium.com/@smallfishbigsea/notes-of-squeezenet-4137d51feef4"
[78] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. (2016). *"Understanding and improving convolutional neural networks via concatenated rectified linear units"*. In Proceedings of the International Conference on Machine Learning (ICML).
[79] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. (2017). *"Mobilenets: Efficient convolutional neural networks for mobile vision applications"*. arXiv preprint arXiv:1704.04861.
[80] https://www.slideshare.net/JinwonLee9/mobilenet-pr044
[81] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. (2018). *"Mobilenetv2: Inverted residuals and linear bottlenecks"*. In CVPR.
[82] "https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html"
[83] Z. Qin, Z. Zhang, X. Chen, C. Wang, and Y. Peng. (2018). *"Fd-mobilenet: Improved mobilenet with a fast downsampling strategy"*. In 2018 25th IEEE International Conference on Image Processing (ICIP), pages 1363–1367, IEEE.
[84] X. Zhang, X. Zhou, M. Lin, and J. Sun. (2017). *"Shufflenet: An extremely efficient convolutional neural network for mobile devices"*. arXiv:1707.01083.
[85] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun. (2018). *"Shufflenetv2: Practical guidelines for efficient cnn architecture design"*. In ECCV.
[86] Seyyed Hossein HasanPour, Mohammad Rouhani, Mohsen Fayyaz, and Mohammad Sabokrou. (2016). *"Lets keep it simple, using simple architectures to outperform deeper and more complex architectures"*. In CoRR, abs/1608.06037.
[87] Hasanpour, Seyyed Hossein and Rouhani, Mohammad and Fayyaz, Mohsen and Sabokrou, Mohammad and Adeli, Ehsan. (2018). *"Towards Principled Design of Deep Convolutional Networks: Introducing SimpNet"*. arXiv preprint arXiv:1802.06205.
[88] Huang, G., Liu, S., van der Maaten, L., Weinberger, K.Q. (2017). *"Condensenet: An efficient densenet using learned group convolutions"*. arXiv preprint arXiv:1711.09224

[89] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. (2016). *"Enet: A deep neural network architecture for real-time semantic segmentation"*. arXiv preprint arXiv:1606.02147.

[90] "https://github.com/luckymouse0/SimpleNet-TF"

[91] Sabour, S., Frosst, N., and Hinton, G. E. (2017). *"Dynamic routing between capsules"*. In Neural Information Processing Systems (NIPS).

[92] Bell, S., Zitnick, C.L., Bala, K., Girshick. (2016). *"Inside-outside net: detecting objects in context with skip pooling and recurrent neural networks"*. In CVPR.

[93] "https://github.com/deepblacksky/capsnet-tensorflow"

[94] Larsson, Gustav, Michael Maire, and Gregory Shakhnarovich. (2016). *"FractalNet: Ultra-Deep Neural Networks without Residuals"*. arXiv preprint arXiv:1605.07648 .

[95] R. Girshick, J. Donahue, T. Darrell, and J. Malik. (2015). *"Region-based convolutional networks for accurate object detection and segmentation"*. TPAMI.

[96] J. Dai, Y. Li, K. He, and J. Sun. (2016). *"R-FCN: Object detection via region-based fully convolutional networks"*. In Neural Information Processing Systems (NIPS).

[97] "https://d2l.ai/chapter_computer-vision/rcnn.html#faster-r-cnn"

[98] "https://towardsdatascience.com/review-r-fcn-positive-sensitive-score-maps-object-detection-91cd2389345c"

[99] K. He, G. Gkioxari, P. Dollar, and R. Girshick. (2017). *"Mask r-cnn"*. arXiv:1703.068707

[100] Wei Liu, Andrew Rabinovich, Alexander C. Berg. (2016). *"ParseNet: Looking Wider to See Better"*. arXiv:1506.04579

[101] "https://medium.com/datadriveninvestor/review-parsenet-looking-wider-to-see-better-semantic-segmentation-aa6b6a380990"

[102] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi. (2015). *"You Only Look Once: Unified, Real-Time Object Detection"*. arXiv:1506.0264

[103] "https://pjreddie.com/yolo/"

[104] J. Redmon and A. Farhadi. (2017). *"YOLO9000: Better, faster, stronger"*. In CVPR.

[105] J. Redmon and A. Farhadi. (2018). *"Yolov3: An incremental improvement"*. arXiv preprint arXiv:1804.02767

[106] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. E. Reed. (2015). *"SSD: single shot multibox detector"*. In CoRR, abs/1512.02325.

[107] "https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab"

[108] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar. (2017). *"Focal loss for dense object detection.* arXiv preprint, arXiv:1708.02002

[109] "https://towardsdatascience.com/review-retinanet-focal-loss-object-detection-38fba6afabe4"

[110] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and ´S. Belongie. (2017). *"Feature pyramid networks for object detection.* In CVPR.

[111] Andrej Karpathy, George Toderici, Sachin Shetty, Tommy Leung, Rahul Sukthankar, and Li FeiFei. (2014). *"Large-scale video classification with convolutional neural networks.* In Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on, pages 1725–1732. IEEE. research.google.com/pubs/archive/42455.pdf.

[112] J. Y. Ng, M. J. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici. (2015). *"Beyond short snippets: Deep networks for video classification"*. In CVPR.

[113] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei L. (2009). *"Imagenet: A large-scale hierarchical image database"*. In CVPR.

[114] Van Gool, L., Williams, C. K. I., Winn, J. and Zisserman, A. (2010). *"The PASCAL Visual Object Classes (VOC) Challenge"*. International Journal of Computer Vision (IJCV), 88(2), 303-338.

[115] J. Xiao, J. Hays, K. Ehinger, A. Oliva, and A. Torralba. (2010). *"SUN Database: Large-scale Scene Recognition from Abbey to Zoo"*. In CPRV.

[116] J. Xiao, K. A. Ehinger, J. Hays, A. Torralba, and A. Oliva. (2010). *"SUN Database: Exploring a Large Collection of Scene Categories"*. In International Journal of Computer Vision (IJCV).

[117] Dua, D. and Graff, C. (2019). *"UCI Machine Learning Repository"*. "http://archive.ics.uci.edu/ml. Irvine", CA: University of California, School of Information and Computer Science.

[118] Griffin, G. Holub, AD. Perona P. *"The Caltech 256.Caltech Technical Report"*.

[119] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, Piotr Dollár. (2014). *"Microsoft COCO: Common Objects in Context"*.

[120] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, van Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, Vittorio Ferrari. (2018). *"The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale"*.

[121] Andreas Geiger andPhilip Lenz and Raquel Urtasun. (2012). *"Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite"*. In Conference on Computer Vision and Pattern Recognition (CVPR).

[122] Andreas Geiger and Philip Lenz and Christoph Stiller and Raquel Urtasun. (2013). *"Vision meets Robotics: The KITTI Dataset"*. International Journal of Robotics Research (IJRR).

[123] Jannik Fritsch and Tobias Kuehnl and Andreas Geiger (2013). *"A New Performance Measure and Evaluation Benchmark for Road Detection Algorithms"*. International Conference on Intelligent Transportation Systems (ITSC).

[124] Moritz Menze and Andreas Geiger. (2015). *"Object Scene Flow for Autonomous Vehicles, Conference on Computer Vision and Pattern Recognition"*. In CVPR.

[125] Alex Krizhevsky. (2009). *"Learning Multiple Layers of Features from Tiny Images"*. Tech Report, 2009.

[126] Grant Van Horn, Oisin Mac Aodha, Yang Song, Yin Cui, Chen Sun, Alex Shepard, Hartwig Adam, Pietro Perona, Serge Belongie. *"The iNaturalist Species Classification and Detection Dataset"*. In CVPR.

[127] B. Zhou, A. Khosla, A. Lapedriza, A. Torralba, and A. Oliva. (2016). *"Places: An image database for deep scene understanding"*. arXiv preprint arXiv:1610.02055.

[128] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). *"Tensorflow: A system for large-scale machine learning"*. Tech. rep., Google Brain, arXiv preprint.

[129] S. Guadarrama, N. Silberman. (2017). *"TensorFlow-Slim: a lightweight library for defining, training and evaluating complex models in TensorFlow"*.

[130] Huang J, Rathod V, Sun C, Zhu M, Korattikara A, Fathi A, Fischer I, Wojna Z, Song Y, Guadarrama S, Murphy K. (2017). *"Speed/accuracy trade-offs for modern convolutional object detectors"*. In CVPR.

[131] "https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md"

[132] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. (2014). *"Caffe: Convolutional architecture for fast feature embedding"*. ArXiv:1408.5093.
[133] "https://github.com/BVLC/caffe/wiki/Model-Zoo"
[134] K. Chatfield, K. Simonyan, A. Vedaldi, A. Zisserman. (2014). *"Return of the Devil in the Details: Delving Deep into Convolutional Nets"*. In BMVC.
[135] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva. (2014). *"Learning Deep Features for Scene Recognition using Places Database"*. In Neural Information Processing Systems (NIPS).
[136] "https://pytorch.org/get-started/locally/"
[137] "https://github.com/pytorch/pytorch"
[138] "https://github.com/caffe2/models"
[139] "https://github.com/caffe2/tutorials''caffe2_tutorials
[140] "https://github.com/hertzdog/corso-deep-learning-noruen/tree/master/notebooks"
[141] "https://github.com/facebookresearch/Detectron/blob/master/MODEL_ZOO.md"
[142] "https://github.com/facebookresearch/Detectron/tree/master"
[143] Ross Girshick and Ilija Radosavovic and Georgia Gkioxari and Piotr Doll and Kaiming He. (2018). *"Detectron"*. In CVPR.
[144] "https://github.com/tensorflow/models/tree/master/research/slim/nets"
[145] "https://github.com/tensorflow/models/tree/master/research/slim#pre-trained-models"
[146] "https://github.com/AlpacaDB/selectivesearch/blob/develop/example/example.py"
[147] W Luo, X Zhao, and TK Kim. (2014). *"Multiple object tracking: A review.* arXiv preprint arXiv:1409.7618.
[148] Keni Bernardin and Rainer Stiefelhagen. (2008). *"Evaluating Multiple Object Tracking Performance: The CLEAR MOT Metrics"*. In EURASIP Journal on Image and Video Processing.
[149] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft (2016). *"Simple online and realtime tracking"*. In IEEE Int. Conf. on Image Proces. pp. 3464–3468.
[150] Kalman, Rudolph Emi. (1960). *"A New Approach to Linear Filtering and Prediction Problems"*. Transactions of the ASME–Journal of Basic Engineering.
[151] "https://it.wikipedia.org/wiki/Peak_signal-to-noise_ratio"
[152] "https://it.ccm.net/contents/708-il-formato-yuv-ycrcb"
[153] "https://it.wikipedia.org/wiki/Sottocampionamento_della_crominanza"
[154] "https://www.compression.ru/video/quality_measure/vqmt_download.html"
[155] "https://imagemagick.org/index.php"