ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO DI UN PROTOTIPO DI VIDEO SORVEGLIANZA INDOOR CON TECNOLOGIE YOLO ED OPENCV

Elaborato in
Programmazione Di Applicazioni Data Intensive

Relatore
Prof. Gianluca Moro

Presentata da
Mattia Pini

Terza Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

Computer Vision
Machine Learning
Convolutiona Neural Network
YOLO
Python

Dedico questo lavoro a chiunque mi sia stato vicino,
e mi abbia aiutato a raggiungere questo travagliato traguardo.

Introduzione

Durante il corso degli ultimi anni, lo studio e lo sviluppo di nuove tecnologie nell'ambito del Machine Learning e in particolare delle Reti Neurali Artificiali, ha permesso la creazione di sofisticati sistemi di elaborazione dati che fino a qualche hanno fa risultavano impossibili per un calcolatore.

La *Computer Vision*, che studia l'elaborazione di immagini digitali e video, ha avuto negli ultimi anni un grande impulso, grazie anche allo sviluppo di schede video sempre più potenti che hanno permesso la crezione sistemi sempre più complessi e performanti.

Questa tesi intende sperimentare l'efficacia di *reti neurali convoluzionali* nello sviluppo di un prototipo di video sorveglianza in ambiente indoor, come un laboratorio informatico, dove è di particolare importanza sapere se un determinato oggetto è ancora presente nell'ambiente e quali persone ci sono state al suo interno.

Il progetto è stato suddiviso in tre macro parti per poi essere unite in un unico componente alla fine dello sviluppo:

- **Riconoscimento di oggetti**: questa sezione rappresenta il modulo dedicato al riconoscimento di oggetti prestabiliti sfruttando una *Convolutional Neural Network*, che risulta essere lo stato-dell'arte per il riconoscimento di oggetti in quanto a velocità e precisione, e del relativo framework per l'addestramento della rete stessa

- **Face Detection**: rappresenta il modulo dedicato al rilevamento della presenza di un volto all'interno di un'immagine

- **Face Recognition**: questo modulo ha il compito, una volta rilevata la presenza di uno o più volti all'interno di un'immagine, di attribuirne l'identità. Al suo interno inoltre troviamo tutti gli strumenti necessari per generare aumentare il dataset e generare un nuovo modello.

La parte finale rappresenta il prototipo di video sorveglianza che unisce le tre componenti sopra elencate per prendere decisioni e salvare eventi importanti come lo spostamento o rimozione di un oggetto dall'ambiente.

I contenuti della tesi sono organizzati nei seguenti capitoli:

- **Le Reti Neurali Artificiali**: Descrive brevemente il funzionamento di un neurone e di come il suo studio abbia portato all'invenzione delle *Reti Neurali Artificiali*. Vengono inoltre descritti i paradigmi di apprendimento e i vari tipi di Reti Neurali Artificiali esistenti.

- **Convolutional Neural Network**: In questo capitolo viene descritto il funzionamento di questo particolare tipo di Rete Neurale e delle sue peculiarità.

- **YOLO: You Only Look Once**: Si descrive brevemente la storia delle Convolutional Neural Network, del progresso che hanno portato all'ambito della *Computer Vision*, per poi entrare più nel profondo nella struttura della tecnologia YOLO.

- **Sviluppo del Progetto**: si spiega dello sviluppo, della sperimentazione e delle problematiche riscontrate durante la realizzazione del progetto; viene inoltre esposto il codice utilizzato e spiegati i dataset utilizzati.

Indice

Introduzione **vii**

1 Le Reti Neurali Artificiali **1**
1.1 Il Percettrone 1
1.2 Deep Learning 2
1.3 Il Concetto di Artificial Neural Network 3
1.4 Apprendimento della rete 4
1.4.1 Apprendimento 4
1.4.2 Gradient Descent 5
1.4.3 Backpropagation 6
1.4.4 Le funzioni di attivazione 7
1.5 Tipi di apprendimento della rete 9
1.5.1 Apprendimento supervisionato 9
1.5.2 Apprendimento non supervisionato 9

2 CNN: Convolutional Neural Network **11**
2.1 Input 11
2.2 Struttura 12
2.2.1 Convolutional Layer 12
2.2.2 Pooling Layer 13
2.2.3 Fully-Connected Layer 14

3 YOLO: You Only Look Once **15**
3.1 Da AlexNet a YOLO 15
3.1.1 AlexNet 15
3.1.2 GoogLeNet 15
3.1.3 ResNet 16
3.2 YOLO 16
3.2.1 YOLOv1 16
3.2.2 YOLOv2 18
3.2.3 YOLOv3 19

4 Sviluppo del progetto **21**
4.1 Prerequisiti del progetto 21
4.2 Riconoscimento oggetti 21
4.2.1 Il modello 21
4.2.2 Test del modello 24
4.3 Riconoscimento facciale 27
4.3.1 Face Detection 27
4.3.2 Face Recognition 29
4.4 Il sistema di sorveglianza 32

5 Conclusioni e sviluppi futuri **35**

Conclusioni e sviluppi futuri **35**

Ringraziamenti **37**

Bibliografia **39**

Elenco delle figure

1.1 Schema computazionale di un Percettrone, come descritto da F. Rosenblatt. 2
1.2 Schema di una semplice rete neurale con un solo strato nascosto 3
1.3 Schematizzazzione di una Feed Forward Network e di una Recurrent Neural Network 4
1.4 Discesa del Gradiente dal punto A a B (Minimo) 5
1.5 A sinistra (a) un fenomeno di Overshooting mentre a destra (b) un apprendimento troppo lento 6
1.6 Grafico della funzione sigmoidale 7
1.7 Grafico della funzione tangente iperbolica 7
1.8 Grafico della funzione ReLU 8
1.9 Differenza tra ReLU e Leaky ReLU 8

2.1 Rappresentazione di un Tensore di una possibile immagine RGB 11
2.2 Schematizzazzione di una semplice Convolutional Neural Network 12
2.3 Esempio di convoluzione con un filtro 3x3 13
2.4 Max Pooling con filtro di dimensione 2x2 e passo = 2 14

3.1 Struttura della rete YOLO nella sua prima versione 17
3.2 La rete YOLOv2 con le migliorie apportate e il suo risultato sul dataset VOC 2007 19
3.3 La rete YOLOv3 allo stato attuale 20

4.1 Output di addestramento del modello dopo 1500 iterazioni . . . 24
4.2 Predizione degli oggetti dal modello 25
4.3 Il riconoscimento facciale distingue il soggetto del training dal fratello 31

Capitolo 1

Le Reti Neurali Artificiali

Nonostante i progressi tecnologici e l'aumento esponenziale di potenza computazionale degli ultimi anni, persistono problemi non banali nella loro risoluzione da parte di una macchina. Una serie di compiti quali possono essere la localizzazione di un oggetto in una scena o l'output di una certa risposta basato su un ragionamento percettivo sono esempi di come un' azione risulti semplice per una persona, ma infinitamente complicata da eseguire per vie convenzionali all'interno di un calcolatore.

1.1 Il Percettrone

Il cervello umano è in grado di elaborare informazioni attraverso un complesso sistema neurobiologico, il cui elemento fondamentale è detto Neurone.
I processi elettrochimici che hanno sede all'interno di questa cellula, processano gli input degli altri neuroni e li modulano per poi inviare questi nuovi segnali alle altre parti del cervello.
Tuttavia un calcolatore non dispone di queste abilità, ma durante il corso degli anni si è cercato di identificare un modello matematico in grado di simularne il comportamento.
Nasce quindi un primo modello computazionale di neurone artificiale quando F. Rosenblatt fornisce il concetto di Perceptron o Percettrone [1]

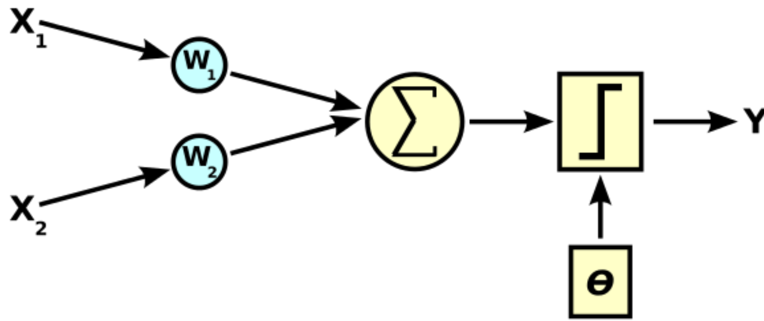

Figura 1.1: Schema computazionale di un Percettrone, come descritto da F. Rosenblatt.

Il Percettrone è un tipo di classificatore binario che associa un insieme di input, ad un output scalare y (di tipo reale) che viene calcolato come:

$$y = \theta(\sum_{i=1}^{n} w_i x_i - b) \tag{1.1}$$

Dove $\sum_{i=1}^{n} w_i x_i - b$ è la somma pesata delle corrispondenti x_i del vettore degli input, ciascuno pesato col il peso w_i e dove θ rappresenta una funzione chiamata Funzione di Attivazione.
Il valore b, detto bias, permette di 'spostare' la funzione di attivazione a destra o sinistra per adeguarsi meglio ai dati in input.
Questo valore può risultare critico per l'apprendimento poichè valori errati potrebbero portare a risultati non coerenti (a seconda di che funzione di attivazione si usi per i neuroni).

1.2 Deep Learning

E' un nuovo approccio al Machine Learning che si basa su un modello a più strati, dove ogni strato rappresenta un livello di astrazione.
Esistono tre tipi di strati:

- strati di basso livello: sono gli strati più vicini all'input, analizzano dati grezzi
- strati intermedi: rielaborano le informazioni estraendo feature sempre più complesse
- strati finali: forniscono l'informazione desiderata ad alto livello

La rappresentazione ottimale dell'output è appreso dagli esempi utilizzati per addestrare il modello, senza il bisogno di dover 'ingegnerizzare' manualmente le feature richieste.

1.3 Il Concetto di Artificial Neural Network

L'architettura di una rete neurale artificiale basilare è derivata da quella del Percettrone. Può essere vista, infatti, come una sequenza di strati interconnessi di neuroni (o Percettroni) dove ogni strato esegue una funzione f sulla somma pesata dei neuroni precedenti e il risultato e' passato allo strato successivo.
Sono il modello più utilizzato nel **Deep Learning**.
Un esempio di una semplice rete neurale è:

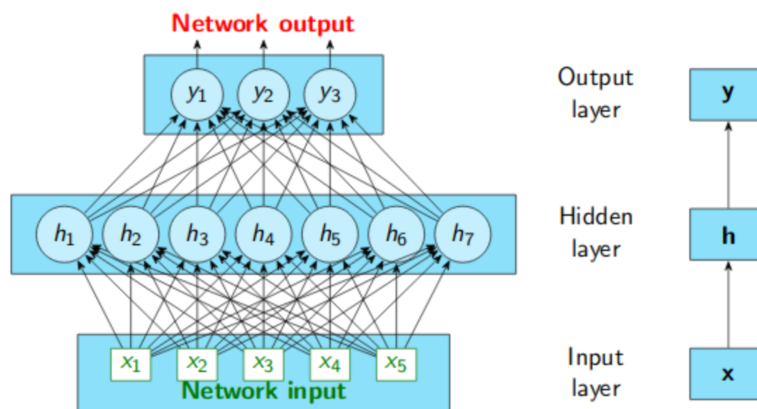

Figura 1.2: Schema di una semplice rete neurale con un solo strato nascosto

Ci sono due categorie principali per suddividere le ANN, esse sono:

- Reti *Feed-Forward* : sono le piu' classiche reti neurali, sono composte da tanti strati (o layer) di nodi completamente connessi.

- Reti *Ricorrenti (RNN)*: sono reti neurali che utilizzano connessioni cicliche per mantenere uno stato attraverso molteplici input e vengono spesso utilizzate per elaborare dati sequenziali.

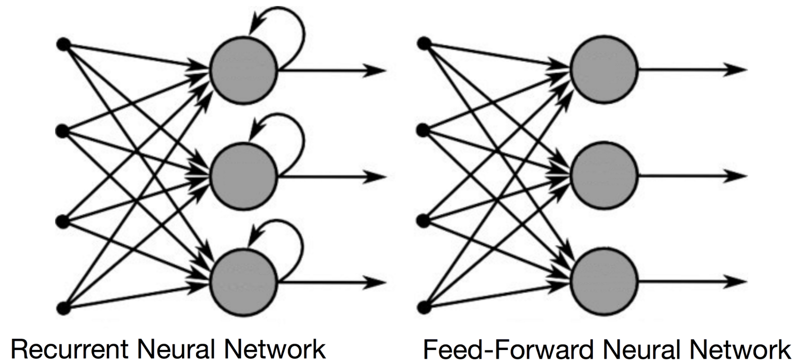

Figura 1.3: Schematizzazzione di una Feed Forward Network e di una Recurrent Neural Network

1.4 Apprendimento della rete

1.4.1 Apprendimento

Ogni rete ha dei parametri costituiti dai pesi delle connessioni tra i neuroni che vengono inizialmente impostati in maniera randomica. L'obiettivo è quello di calibrare i pesi per minimizzare la funzione di perdita (o *Loss Function*). Questa funzione determina di quanto si discostano i valori predetti dal sistema dai valori reali chiamati *Targets* o *Ground Truth*.
Una delle funzioni di perdita più utilizzate è la *Mean Squared Error (MSE)* che è definita dalla seguente formula:

$$MSE = \frac{1}{n}\Big(\sum_{i=1}^{n} y_i - \hat{y}_i\Big)^2 \tag{1.2}$$

Dove n è il numero dei campioni, y_i è il valore reale i-esimo e $\hat{y}_i$ è il valore stimato dal modello.
Prediamo in considerazione ora la funzione appena definita ed utilizziamola come nostra funzione di perdita:

$$Loss(\omega) = \frac{1}{n}(\sum_{i=1}^{n} y_i - \hat{y}_i)^2 \tag{1.3}$$

Per quanto definito prima nel percettrone [1.1], possiamo sostituirlo nella formula di perdita al posto di $\hat{y}_i$ ottenendo:

$$Loss(\omega) = \frac{1}{n}\Big(\sum_{i=1}^{n} y_i - \theta(\sum_{i=1}^{n} w_i m_i - b)\Big)^2 \quad (1.4)$$

Questo è permesso grazie a un processo di inferenza dei risultati chiamato *Forward Propagation.*
L'obiettivo ora è quello di minimizzare la funzione di perdita, ovvero individuare quei ω tali per cui $Loss(\omega)$ sia minimo.
Per effettuare questo calcolo si usa il metodo del **Gradient Descent** o **Discesa del Gradiente**

1.4.2 Gradient Descent

Data un funzione f a più variabili, il gradiente ∇f è definito come il vettore delle derivate parziali di f per per ciascuna delle sue variabili.
Fissato un punto x, il gradiente ci indica l'inclinazione della curva nel punto analizzato, permettendo di sapere quindi se la funzione cresce maggiormente, e scegliere di conseguenza in quale direzione 'muoversi' sulla funzione per minimizzarla.
Per visualizzare questo concetto, immaginiamo un sistema con soli due pesi α e β rispettivamente sull'asse x e y.
Dato questo sistema, i valori della funzione di perdita ci definiscono l'asse z come definito nell'esempio sottostante

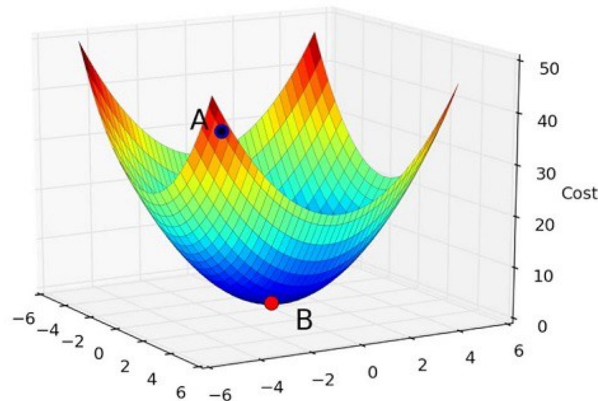

Figura 1.4: Discesa del Gradiente dal punto A a B (Minimo)

L'obiettivo quindi è quello di trovare una combinazione di pesi tali per cui la funzione di perdita sia minima, muovendosi in direzione opposta al gradiente.

1.4.3 Backpropagation

L'algoritmo di Backpropagation è un metodo introdotto negli anni '70 ma la sua importanza fu nota solo dal 1986 dopo la pubblicazione di un importante articolo da parte di David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams [2] che ha dato il via alla rivoluzione delle reti neurali.
Questo metodo è fondalmente diviso in due fasi:
La prima detta anche *Forward Propagation* è la parte dove il dato in input viene 'passato' da un layer al successivo per infine generare un risultato, mentre la seconda è la parte dove i pesi vengono aggiornati affinche' la rete approssimi in maniera sempre più accurata la funzione desiderata. L'aggiornamento dei pesi si può definire come:

$$W_i = W_{i-1} - \alpha \frac{d}{d\omega} Loss(\omega) \tag{1.5}$$

I pesi al passo i-esimo vengono aggiornati sottrandedo al passo i-1 la derivata dell'errore stimato moltiplicata per un fattore α chiamato *Learning Rate*.
Il *Learning Rate* regola la velocità con cui la rete 'apprende', ovvero abbandonare vecchie conoscenze per quelle nuove. Se la velocita di apprendimento è troppo elevata si rischia di incorrere nel fenomeno di *Overshooting*, ovvero specializzarsi troppo sul nuovo input considerato e non tenere considerazione dei precedenti, generando un salto dal punto di minimo locale.
Al contrario se il *Learning Rate* è troppo basso, la convergenza a un minimo globale potrebbe richiedere molto tempo ed è maggiore la percentuale di bloccarsi su un minimo locale.

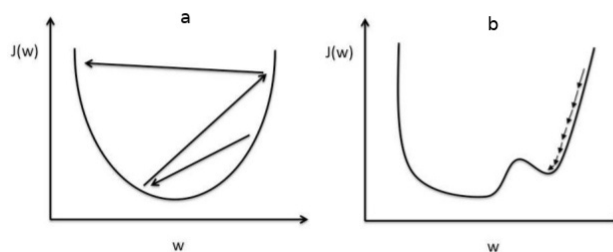

Figura 1.5: A sinistra (a) un fenomeno di Overshooting mentre a destra (b) un apprendimento troppo lento

1.4.4 Le funzioni di attivazione

L'output di un nodo (o percettrone) è modulato dalla funzione di attivazione θ (come definito in 1.1).
Contrariamente al passato dove veniva utilizzata una funzione di attivazione a soglia fissa (`Binary Step`), oggi si prediligono principalmente funzioni di attivazioni non lineari.
Vediamo di seguito le principali funzioni di attivazione:

- $Logistic\ function$ ($StepSoft$):

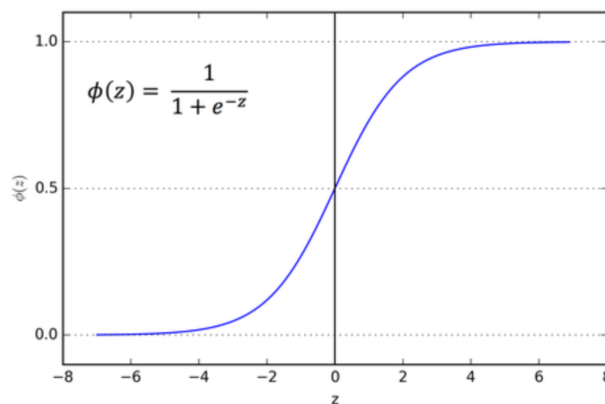

Figura 1.6: Grafico della funzione sigmoidale

- $Tanh\ Function$:

$$f(x) = tanh(x) = \frac{2}{1+e^{-2x}} - 1 \tag{1.6}$$

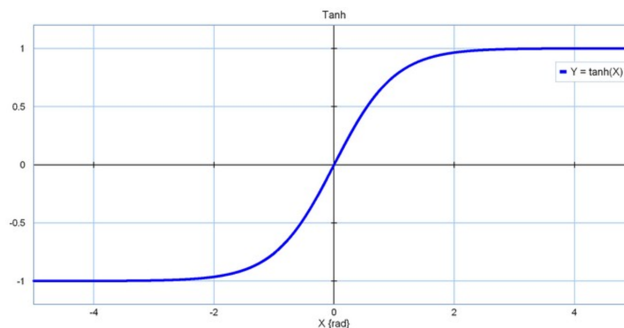

Figura 1.7: Grafico della funzione tangente iperbolica

- *ReLU* (*Rectified Linear Unit*):

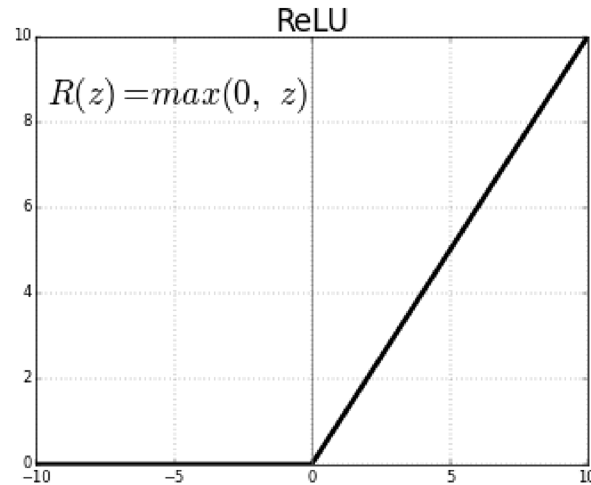

Figura 1.8: Grafico della funzione ReLU

Le funzioni di attivazione ReLU a differenza della sigmoidale e tangente iperbolica , non soffrono del problema del **Vanishing Gradient** ovvero la diminuzione degli effetti di 'apprendimento' quando ci si avvicina agli estremi della funzione di attivazione.
Tuttavia nemmeno ReLU è immune da problematiche: qualsiasi input negativo diventa immediatamente zero, precludendo cosi alla rete di poter addestrarsi correttamente su tutti i dati.
Per risolvere questo problema viene introdotta la Leaky ReLU, che viene utilizzata come funzione di attivazione della rete YOLO dai layer di convoluzione.

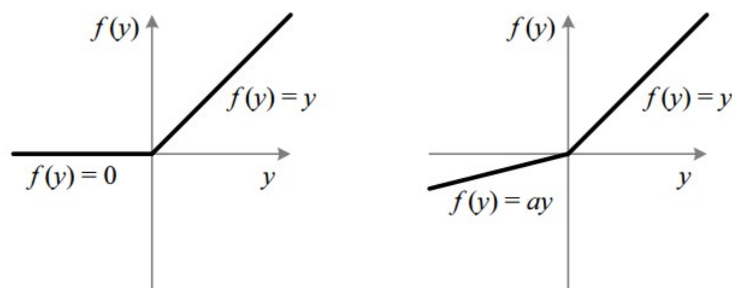

Figura 1.9: Differenza tra ReLU e Leaky ReLU

1.5 Tipi di apprendimento della rete

1.5.1 Apprendimento supervisionato

La gran parte dei problemi risolvibili con il Machine Learning sono di tipo supervisionato.
Questi sono problemi in cui si dispone di un Dataset, di conseguenza gli output sono noti, perciò si dispone delle necessarie coppie di dati X (ingressi) e Y (uscita).
L'obiettivo quindi è quello di approssimare la funzione che mappa gli input X agli output Y in modo che alla vista di un nuovo campione, la rete sia in grado di predire un output Y significativo.
I problemi di tipo supervisionato sono divisibili in due categorie:

- `Classificazione`: Predizione di un output di tipo categorico, per esempio classificazione del contenuto di un'immagine come contenente 'gatto' o 'cane'.
- `Regressione`: Stimare una variabile reale Y in funzione di 1 o più parametri in ingresso

1.5.2 Apprendimento non supervisionato

L'apprendimento si dice non supervisionato se si dispone dei soli input X in ingresso senza corrispondenti variabili di output.
L'obiettivo in questi casi è quello di capire la struttura dei dati, indipendentemente dalla conoscenza delle variabili di output.
I problemi di tipo non supervisionato possono essere classificati in due aree:

- **Clustering**: Problemi in cui si vuole scoprire come formare dei gruppi a seconda dei dati di input.
- **Associazione**:Un problema di associazione si ha quando si vuole scoprire una relazione che descrive grandi porzioni di dati, come per esempio: 'Le persone che guardano questi programmi TV spesso guardano anche questi altri programmi'.

Capitolo 2

CNN: Convolutional Neural Network

In questo capitolo si discuterà dell'architettura di questa rete neurale artificiale e di come riesca efficacemente a estrarre feature da immagini.

2.1 Input

Come ben sappiamo un calcolatore non è in grado di visualizzare immagini come fa un essere umano, quindi un'immagine deve essere convertita in una serie di matrice di dimensione MxN pixel, di profondità variabile in base al numero di canali che possiede (nel caso di RGB i canali saranno tre, nel caso di un'immagine a scala di grigi è sufficiente un solo canale).
Questo insieme di matrici è anche chiamato Tensore.

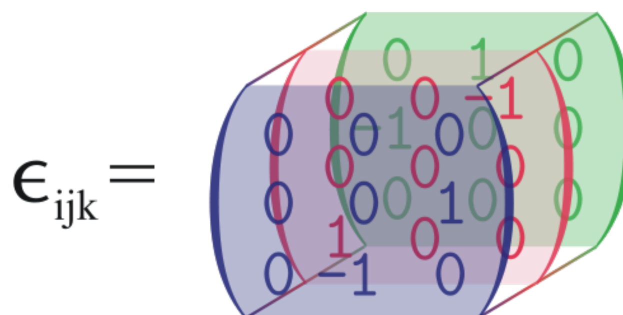

Figura 2.1: Rappresentazione di un Tensore di una possibile immagine RGB

Il contenuto delle matrici è un valore numerico che sta in un intervallo da 0 a 255, ed indica la predominanza cromatica di quel singolo canale. E' buona

norma all'interno di una rete neurale effettuare un processo di *Normalizzazione* affinchè l'intervallo da considerare sia più piccolo.

2.2 Struttura

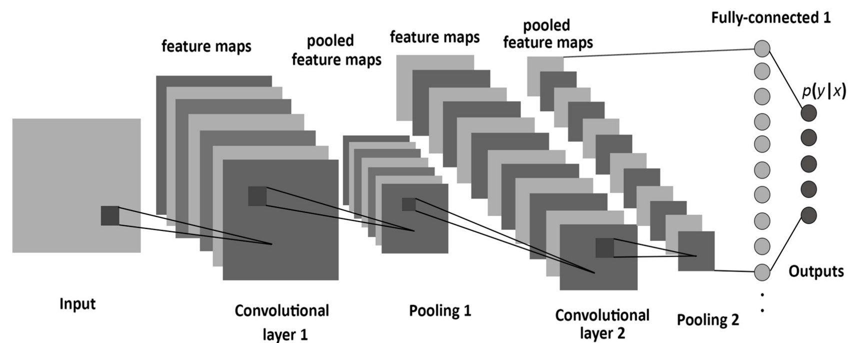

Figura 2.2: Schematizzazzione di una semplice Convolutional Neural Network

Una *Convolutional Neural Network* è composta da un serie non precisamente definita di layer che si alternano tra *Convolutional Layer* , *Pooling Layer* e *Fully-Connected Layer* (questi ultimi generalmente usati come strati finali della rete).

2.2.1 Convolutional Layer

Il *Convolutional Layer* si occupa di gestire la convoluzione facendo scorrere un $Filtro$ sul $Tensore$; il $Filtro$ è un tensore di dimensione m x n di profondita pari all'immagine in ingresso. Il risultato della convoluzione è il prodotto scalare dei valori originali dei pixel con i valori presenti nel filtro.

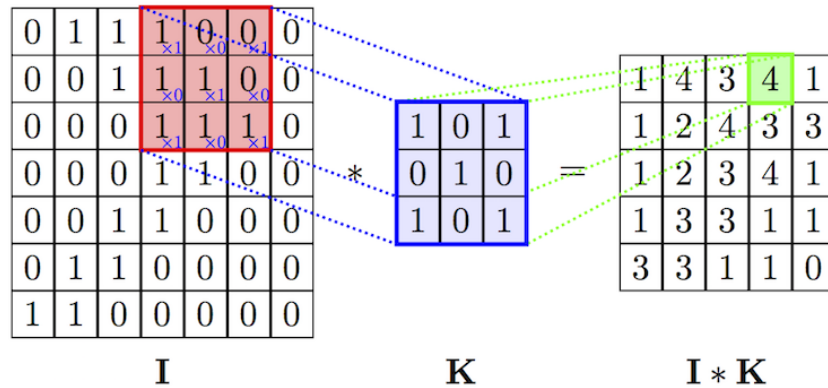

Figura 2.3: Esempio di convoluzione con un filtro 3x3

Questo processo viene eseguito per tutta l'immagine e il *Tensore* in uscita viene chiamato *Activation Map* o *Feature Map* e ci indica la presenza o meno di una determinata feature all'interno del tensore.
L'individuazione di feature complesse cresce mano a mano che la profondità della rete aumenta: feature più semplici (curve, linee) vengono riconosciuti dai primi layer di convoluzione, mentre feature più complesse (volti, oggetti) vengono identificati da layer più in profondità.
Alla fine di un *Convolutional Layer* si può effettuare un'operazione di Pooling gestita dal *Pooling Layer*.

2.2.2 Pooling Layer

L'operazione di Pooling consiste nella ridimensionalità dell'output del layer precedente, in modo da avere una rappresentazione più compatta delle feature estratte. A differenza del *Convolutional Layer*, il *Pooling Layer* non esegue la combinazione lineare, ma fa "scorrere" un *filtro* di dimensione z x z seguendo un passo k sul Tensore analizzandole una regione per volta.

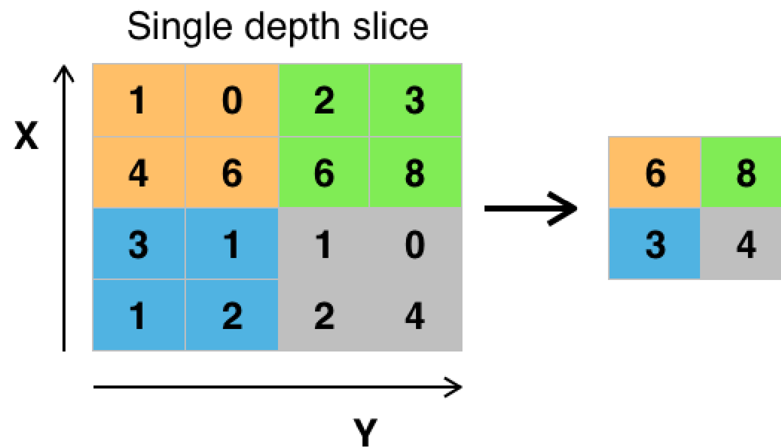

Figura 2.4: Max Pooling con filtro di dimensione 2x2 e passo = 2

Esistono diverse metodologie di Pooling tra cui:

- **Avg Pooling**:Viene restituita una media della regione esaminata
- **Max Pooling**:Viene restituito il valore massimo della regione esaminata

2.2.3 Fully-Connected Layer

Sono i Layer finali della rete e il loro compito è quello di creare la rappresentazione finale del dato in uscita. Questi layer sono spesso interconnessi completamente tra di loro e prendono il nome di *Dense Layer*, dove ogni nodo di un layer è connesso con tutti i nodi del layer successivo.
Come ultimo strato, in base al problema da risolvere, può essere aggiunta una funzione di attivazione; nel caso di un problema di classificazione, per esempio, si userà una funzione probabilistica come la *SoftMax*.

Capitolo 3

YOLO: You Only Look Once

In questo capitolo verrà affrontato il discorso della rivoluzione che le *Convolutional Neural Network* hanno portato nel mondo della Computer Vision e l'importate contributo che la rete **YOLO** sta portando a quest'ultimo per il problema *Object Detection*, nonchè la rete utilizzata in questa tesi.

3.1 Da AlexNet a YOLO

3.1.1 AlexNet

Fino al 2012, il problema della classificazione degli oggetti all'interno di un'immagine era stata affrontata con soluzioni diverse dalle *Convolutional Neural Network*.
La loro potenza si è mostrata all'annuale competizione ILSVRC (ImageNet Large Scale Visual Recognition Competition) con **AlexNet**[3], la prima *CNN* a fare uso di GPU e vincere una competizione di Image-Classification e ad ottenere un ratio di errore top-5 di 15.3%, migliorando di 10 punti percentuali il concorrente che non faceva uso di *CNN*.
Questo incredibile risultato portò molta attenzione, sia da parte dei ricercatori che da parte delle aziende, sul tema delle *CNN* accoppiato alle GPU.

3.1.2 GoogLeNet

Nel 2014 infatti Google portò alla competizione ILSVRC la sua versione di *CNN* chiamata *GoogLeNet*[4]: questa rete riuscì a ottenere un ratio di errore top-5 di 6.67% migliorando di tantissimo il risultato ottenuto da *AlexNet* nel 2012.
La vera innovazione di *GoogLeNet* fu il modulo *Inception*[4]: questo modulo

si basa su diversi strati di convoluzione molto piccoli in modo da avere un numero molto inferiore di parametri; infatti *GoogLeNet* aveva solo 4 milioni di parametri contro i 60 milioni di *AlexNet*.

3.1.3 ResNet

Il 2015 vede un'altra grande innovazione al ILSVRC, ovvero l'introduzione di un nuovo tipo di rete neurale chiamato *ResNet* o *Residual Neural Network*. Questa rete ottenne un ratio di errore top-5 di 3.57%, segnando la svolta, riuscendo a battere le performance umane sul dataset della competizione.
La novità di questa rete è stata l'introduzione della possibilità di 'saltare' dei layer durante la fase di addestramento per accelerarne il processo e risolvere il problema del *vanishing gradient*.

3.2 YOLO

Tutte queste architetture viste fin'ora sono state utilizzate per il problema della classificazione di immagini, ovvero data un'immagine identificarne il suo contenuto.
Tuttavia per il problema del *Object Detection*, ovvero il riconoscimento di un oggetto dentro a un'immagine e la sua posizione, tutte le reti viste fin'ora sono inadeguate poichè i tempi di calcolo sono troppo elevati per garantire una risposta in tempi molto brevi.

3.2.1 YOLOv1

La rete YOLO si presenta come soluzione a questo problema. Nella prima versione[6], nel suo modello base, riesce a riconoscere oggetti da sorgente video a 45 *FPS (Frames Per Second)*, ottenendo un *mAP (mean Average Precision)* doppio rispetto alle altre reti come Fast R-CNN[7].
Il problema con le R-CNN è che queste ultime dividono l'immagine in tante regioni indicate con *ROI (Region of Interest)* e di queste si verifica, regione per regione, se all'interno di ognuna ci sia o meno una feature, impiegando ben 47 secondi (nella prima versione) per l'analisi di una singola immagine.
In YOLO invece l'analisi viene fatta su tutta l'immagine; questo tipo di architettura viene chiamata *Single Shot Detection*.
YOLO divide l'intera immagine in una griglia di dimensione MxM; se il centro di un oggetto cade in una cella della griglia, allora quella cella sarà la responsabile per il rilevamento dell'oggetto. Ogni cella può prevedere un solo oggetto per volta e un numero limitato (B) di *Bounding Boxes*. Questi ultimi non sono altro che 'rettangoli' che circondano l'oggetto interessato per

evidenziarlo rispetto al resto dell'immagine.
Ogni *Bounding Box* contiene 5 elementi: x, y, w, h e la confidenza dell'oggetto rilevato; x e y sono l'offset all'interno della griglia interessata rispetto ai suoi bordi, w e h sono la dimensione relative del *Bounding Box* rispetto all'intera immagine mentre la confidenza è data da quanto il modello è 'convinto' che all'interno del Box ci sia un oggetto e di quanto e di quanto è accurato rispetto all'oggetto stesso. Viene definita inoltre la probabilità condizionale della classe di appartenenza (*conditional class probability*) che ci indica la probabilità che un oggetto appartenga a una determinata classe.
Il risultato finale è un tensore di dimensione (M, M, $B*5+C$) dove C sono il numero di classi su cui è stato addestrato il modello.
L'architettura della rete è una *CNN* composta da 24 layer seguiti da 2 *Fully-Connected Layer* come mostrato in foto:

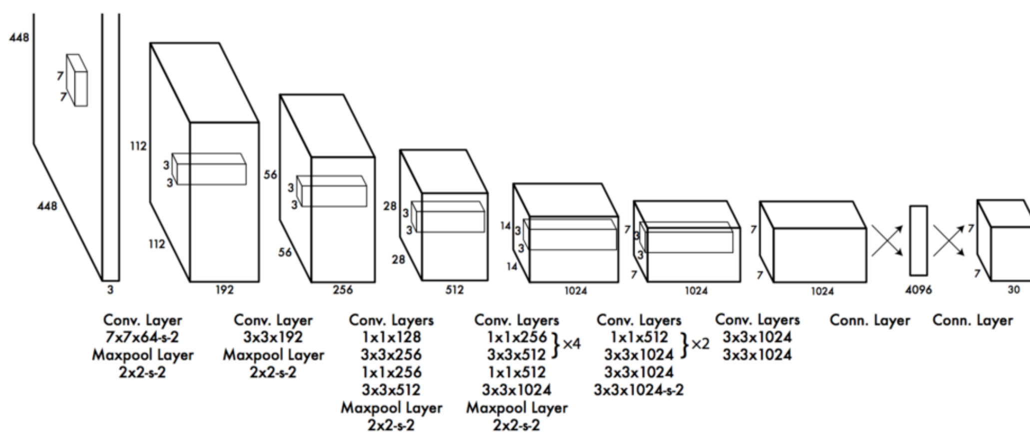

Figura 3.1: Struttura della rete YOLO nella sua prima versione

Esiste anche una versione semplificata della rete che fa uso solo di 9 layer di convoluzione, ma che riesce a raggiungere velocità molto più elevate perdendo di pochi punti percentuali sulla precisione. YOLO fa anche uso della tecnica Non-Maxima Suppression per evitare che lo stesso oggetto venga rilevato due volte.
Tuttavia la rete YOLO non è immune da problemi: infatti non riesce a rilevare correttamente oggetti piccoli molto vicini come per esempio uno stormo di uccelli. In ogni caso il risultato è nettamente migliore rispetto ai suoi concorrenti perchè riesce a generalizzare meglio e ad adattarsi meglio in ambienti diversi (i.e. immagini reali contro dipinti).

3.2.2 YOLOv2

Quasi un anno dopo dall'uscita della prima versione, venne rilasciata una nuova versione della rete YOLO[8] che apportava le seguenti migliorie:

- **Batch Normalization**: e' stato introdotto nei layer di convoluzione, aumentando di 2 punti percentuale di mAP eliminando il dropout[9] dalla rete.

- **Classificatori a risoluzione maggiore**: la prima versione di YOLO riduceva la dimensione dell'immagine per l'addestramento del classificatore a 224x224 pixel seguito da immagini al doppio della dimensione per il rilevamento. Nella nuova versione, durante l'addestramento il classificatore viene sempre addestrato con immagini 224x224, ma poi viene fatta un'operazione di 'tuning' con immagini di dimensione 448x448 in modo da aggiustare i pesi per lavorare a risoluzione maggiori.

- **Anchor Boxes**: è stato rimosso l'ultima parte della rete, i due layer FC (*Fully Connected*) che si occupavano del prevedere la posizione dei *Bounding Boxes*, in favore degli *Anchor Boxes*. Questo è dato dal fatto che non tutti gli oggetti hanno la stessa forma, e quindi anchor box di dimensione differenti si specializzano nel riconoscimento di oggetti differenti. Inoltre è stata spostata la predizione della classe dell'oggetto dalla cella della griglia al layer opportuno.

- **Diminuzione dell'immagine in input**: si è passati da un input a 448x448 a 416x416 in modo da avere un solo centro dell'immagine. Infatti si è assunto che oggetti molto grandi tendano ad occupare il centro dell'immagine quindi in questa maniera la predizione avviene più velocemente perchè il numero di celle da analizzare è inferiore.

- **Dimension Cluster e Direct Location prediction** :per individuare il numero di Bounding boxes migliore è stato usato l'algoritmo di K-mean clustering che ha portato al miglior risultato con $k = 5$; inoltre ora la posizione del Bounding Box viene stimata in riferimento a quella dell'Anchor Box nel quale è stato identificato l'oggetto interessato.

- **Fine-Grained Features**: sfruttando l'approccio passthrough ovvero di collegare l'output di un layer di uno strato n-esimo con quelli di uno a un livello inferiore, si riesce a ritornare a uno stato dell'immagine meno deteriorata (per via dei vari layer di convoluzione) cosi da poter distinguere anche oggetti più piccoli.

- **Multi-scale Training**: non essendoci più gli ultimi due *Fully Connected layer*, non abbiamo più vincoli sulla dimensione dell'immagine. Quindi, ogni 10 epoch, la rete durante la fase di training prende in considerazione immagini di dimensione diversa dalle precedenti per forzare la rete ad adattarsi ad immagini a dimensioni diverse.

Tutte le nuove migliorie hanno portato a un grosso incremento delle prestazioni e sulla precisione, come si può notare da questo schema:

	YOLO								YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Figura 3.2: La rete YOLOv2 con le migliorie apportate e il suo risultato sul dataset VOC 2007

Un altro importante cambiamento è stato il passaggio da un framework basato su una variazione di GoogLeNet a Darknet, un framework open-source per Reti Neurali scritto in C e CUDA, ottimizzato quindi per GPU NVIDIA. Per diverso tempo YOLOv2 è stato lo stato dell'arte nel settore del *Object Detection* e *Object Classification* riuscendo a ottenere prestazioni fino a quel momento mai raggiunte.

3.2.3 YOLOv3

Con il continuo sviluppo di nuove tecniche per migliorare i sistemi di riconoscimento e rilevamento oggetti, ben presto YOLOv2 fu surclassato da altre reti come RetinaNet[11] e Light-Head R-CNN[12] riuscendo a superarlo sia in precisione che in velocità.
I nuovi metodi includono l'utilizzo di *Residual block* introdotti da ResNet, l'utilizzo di *Upsampling* ovvero di portare l'immagine da dimensioni inferiori a dimensioni maggiori e l'utilizzo di *Skip Connections*, ovvero la possibilità di portare l'output di un layer anche ad altri layer successivi e non solo a quello

immediatamente dopo.
YOLOv3 implementa tutte queste nuove tecniche, dimostrando nuovemente la sua potenzialità e la potenza del framework Darknet.
Rispetto alla versione precedente cambia anche la sua struttura: YOLOv2 si basava su Darknet-19 ovvero una rete fatta da 19 layer di convoluzione più altri 11 per il rilevamento d'oggetti con un totale di 30 layer; ora YOLO si basa su un nuovo tipo di rete chiamato Darknet53 (analogamente al predecessore, 53 indica i layer di convoluzione) con un totale di 106 layer, permettendo un miglioramento generale delle precisione soprattutto sugli oggetti di piccole dimensioni.
Ecco l'attuale struttura della rete:

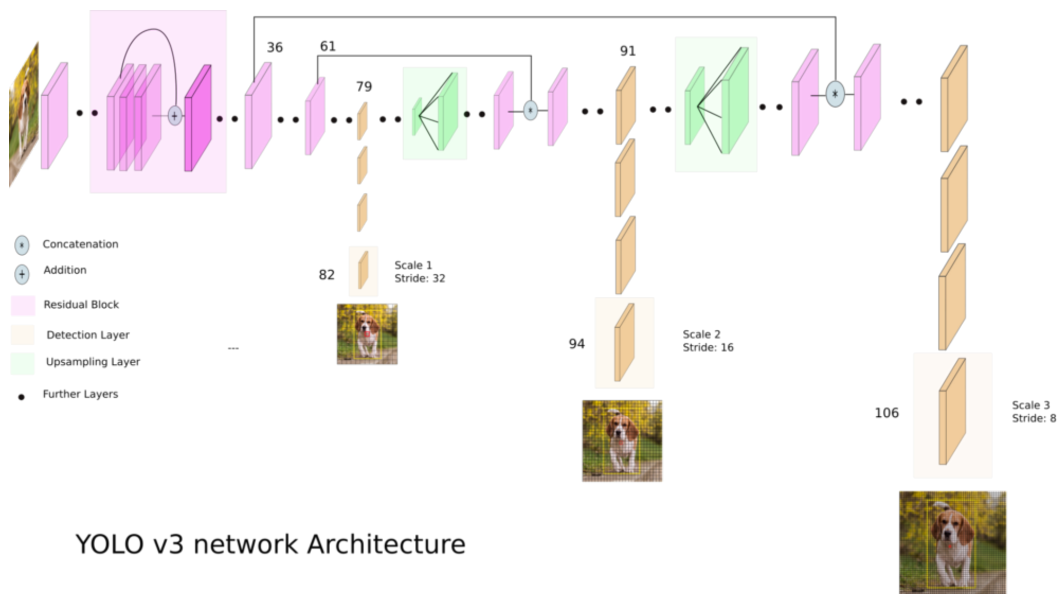

Figura 3.3: La rete YOLOv3 allo stato attuale

L'introduzione di tutti questi layer ha fatto si che la rete ora sia più lenta rispetto alla sua versione precedente (33 FPS su GPU Pascal Titan X, la metà della versione precedente) però guadagnandone in precisione, riuscendo a superare le concorrenti RetinaNet e SSD su una metrica mAP_{50} ovvero con un IOU di 0.5.
Se aumentiamo però la precisione, ovvero usiamo un IOU di 0.75 notiamo che i risultati di YOLOv3 non riescono a equiparare quelli ottenuti da RetinaNet, ma riescano a essere più veloci, dimostrando un trade-off tra precisione e velocità di rilevamento.

Capitolo 4

Sviluppo del progetto

In questo capitolo si esporrà lo sviluppo del progetto e le fasi che lo hanno composto, spiegando le vari componenti utilizzati e il risultato finale prodotto e di come sono stati usate le tecnologie precedenti all'interno del progetto.

4.1 Prerequisiti del progetto

Il progetto si basa su in buona parte sul framework Darknet, che può non essere installato se non si vuole addestrare un nuovo modello per rilevare oggetti di interesse diversi.
Tuttavia è necessaria una libreria dinamica della rete Darknet per il wrapper in Python che racchiude tutte le funzionalità necessarie per il funzionamento del progetto, di cui una versione è distribuita assieme a questo progetto.
Il progetto è stato scritto utilizzando il linguaggio Python3 alla versione 3.6 e facendo uso di diverse librerie tra cui viene evidenziata *OpenCV*, indispensabile per il progetto, che permette l'elaborazione di immagini e video in maniera performante.

4.2 Riconoscimento oggetti

Per il riconoscimento e il rilevamento degli oggetti è stata usata la rete YOLO, iniziando le sperimentazioni con YOLOv2 e in seguito a test e valutazioni si è scelto per YOLOv3.

4.2.1 Il modello

Per poter riconoscere determinati classi di oggetti, è necessario prima addestrare un modello che riesca ad essere il più preciso possibile.

Per addestrare il modello si e' utilizzato il framework Darknet il quale ha bisogno di diversi parametri per poter funzionare:

- **File di configurazione**:è quello che ci permette di indicare la struttura della rete ovvero il numero e i tipi di layer e come sono collegati tra loro, il numero di classi del dataset selezionato e il numero di filtri di determinati layer di convoluzione che dipendono dalla seguente formula $filters = (numClasses + 5) * 3$.
  Il file di configurazione dopo le opportune modifiche si presenta così (una porzione, altrimenti sarebbe troppo lungo):

```
[net]
# Training
batch=64
subdivisions=16
width=608
height=608
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
burn_in=1000
max_batches = 500200
policy=steps
steps=4000,4500
scales=.1,.1

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

#[OTHER LAYERS]
```

  Qui di seguito vengono elencati i significati dei parametri più importanti:

- **[net]**:rappresenta l'inizio della nostra rete
- **width e height**: rappresentano la dimensione dell'ingresso, che deve essere un multiplo di 32.
- **batch e subdivision**: indica la porzione di dataset da considerare a ogni iterazione di training
- **momentum e decay**: sono parametri noti nel Deep Learning; il primo è un parametro per l'algoritmo di Gradient descent che permette una convergenza più veloce, mentre decay è di quanto diminuisce il learning rate dopo un certo intervallo di tempo, per far si che il learning rate non sia costante
- **[convolutional]**: rappresenta un layer di convoluzione seguito dai suo parametri di configurazione; ovviamente ci sono anche altri layer come **[shortcut]** che rappresentano gli *skip-connecction* come introdotto da ResNet

• **File di data**: questo file molto più semplice del precedente contiene i path dei file da cui la rete andrà a caricare le immagini, il numero delle classi da allenare, il file contenente le label ordinate in base all'id della classe e una cartella dove salvare il modello.
Per esempio un file di data può essere scritto cosi:

```
classes=2
train = data/test/train.txt
valid = data/test/test.txt
names = data/test/test.names
backup = backup/
```

• **Pesi iniziali della rete**: molte volte per inizializzare una rete si utilizzano dei valori casuali che poi vanno via a via ottimizzati.
Tuttavia Darknet da la possibilità di avere un set di pesi già ottimizzato in parte su ImageNet, diminuendo di molto i tempi di convergenza.
Una peculiarità della rete è che se l'addestramento viene fermato, è possibile riprenderlo impostando come pesi quelli calcolati dalla rete e non più quelli preconfigurati.

Prima di procedere con l'addestramento è stato necessario preparare le immagini con i relativi *Bounding Boxes.*
I dati necessari per il training sono stati reperiti su ImageNet, tuttavia è stato necessario un lavoro di conversione dei file di *Bounding Boxes* poichè ImageNet li fornisce in formato xml e non compatibile con la rete YOLO.
Appositi script in Python sono stati scritti per facilitare la preparazione dei

dati e verranno resi pubblici prossimamente.
Una volta impostati tutti i parametri e preparato le immagini, si può dare il via all'addestramento, il quale ha richiesto diverso tempo a causa di un calcolatore non troppo performante (GTX 1070 8GB VRAM, 8GB RAM DDR3,CPU Intel i5-3570k).
Possiamo notare dall'immagine che dopo quasi due ore sono state effettuate 1500 iterazioni e che la *Loss Function* ha gia un valore sotto a 0.5 (minore è il valore, migliore è la precisione del modello), e di come riesca già a stimare un buon IOU 0.5 e 0.75.

```
File  Modifica  Schede  Aiuto
75R: -nan,  count: 0
Region 82 Avg IOU: 0.786222, Class: 0.999184, Obj: 0.962899, No Obj: 0.001214, .
5R: 1.000000, .75R: 0.666667,  count: 3
Region 94 Avg IOU: 0.555070, Class: 0.990648, Obj: 0.744017, No Obj: 0.000124, .
5R: 0.500000, .75R: 0.500000,  count: 2
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .
75R: -nan,  count: 0
Region 82 Avg IOU: 0.750011, Class: 0.999348, Obj: 0.603024, No Obj: 0.001630, .
5R: 1.000000, .75R: 0.750000,  count: 4
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000004, .5R: -nan, .7
5R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .
75R: -nan,  count: 0
Region 82 Avg IOU: 0.704690, Class: 0.993996, Obj: 0.629975, No Obj: 0.001671, .
5R: 1.000000, .75R: 0.200000,  count: 5
Region 94 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000004, .5R: -nan, .7
5R: -nan,  count: 0
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .
75R: -nan,  count: 0
1520: 0.275770, 0.277114 avg, 0.001000 rate, 8.852706 seconds, 97280 images
```

Figura 4.1: Output di addestramento del modello dopo 1500 iterazioni

4.2.2 Test del modello

Per testare la validità del modello appena addestrato, è stato usato nuovamente Darknet solo in fase di test per vedere se effettivamente delle classi di oggetti presenti nel dataset venivano riconosciute o meno, e come raffigurato dalla foto gli oggetti vengono riconosciuti ed evidenziati dai relativi Bounding Box.

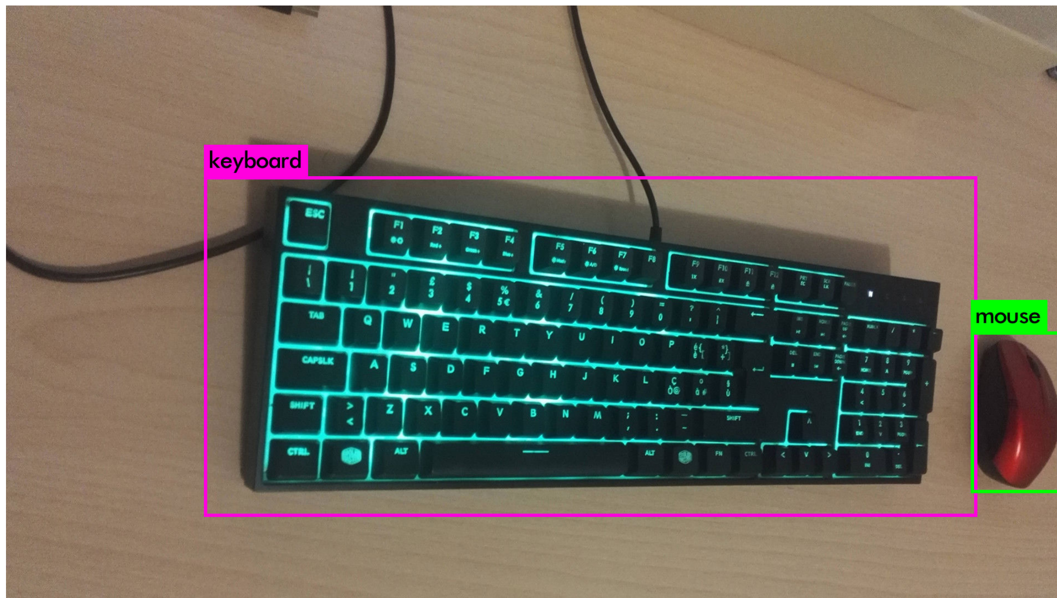

Figura 4.2: Predizione degli oggetti dal modello

Inoltre il modello è stato validato attraverso le API fornite da COCO (Common Object in Context) e usando la metrica VOC PASCAL, ovvero con IoU >= 0.5, è stato ottenuto un mAP di 50.69%, dimostrando che il modello riesce a riconoscere in maniera accurata le classi su cui è stato addestrato.
Una volta testato e dimostrato che il modello è funzionante, è stato utilizzato nel progetto.
Per poter utilizzare il modello in maniera più semplice e per adattare YOLO al fine della tesi, è stato scritto un apposito Wrapper in Python3 (prendendo spunto da quello proposto dal creatore di YOLO) che inizializza tutte le strutture dati necessarie per la rete riuscendo a mantenere buone prestazioni.
Qui di seguito una porzione del wrapper con le strutture dati principali e i metodi più utilizzati:

```
import ctypes
import math
import random
import os
# Darknet dinamic library
lib_name = 'libdarknet.so'
lib_path = os.path.join(os.path.dirname(os.path.abspath(__file__)),
    lib_name)
if os.path.exists(lib_path):
    pass
elif os.path.exists(os.path.join(os.getcwd(), lib_name)):
```

```
    lib_path = os.path.join(os.getcwd(), lib_name)
else:
    print(f'library: {lib_name} can not be found!')

class BOX(ctypes.Structure):
    _fields_ = [("x", ctypes.c_float),
                ("y", ctypes.c_float),
                ("w", ctypes.c_float),
                ("h", ctypes.c_float)]

class DETECTION(ctypes.Structure):
    _fields_ = [("bbox", BOX),
                ("classes", ctypes.c_int),
                ("prob", ctypes.POINTER(ctypes.c_float)),
                ("mask", ctypes.POINTER(ctypes.c_float)),
                ("objectness", ctypes.c_float),
                ("sort_class", ctypes.c_int)]

class IMAGE(ctypes.Structure):
    _fields_ = [("w", ctypes.c_int),
                ("h", ctypes.c_int),
                ("c", ctypes.c_int),
                ("data", ctypes.POINTER(ctypes.c_float))]

class METADATA(ctypes.Structure):
    _fields_ = [("classes", ctypes.c_int),
                ("names", ctypes.POINTER(ctypes.c_char_p))]

#load the dinamic library
lib = ctypes.CDLL(lib_path, ctypes.RTLD_LOCAL)
.
.
.
```

Questo metodo richiama la funzione di Darknet per trovare i Boundig Boxes di un'immagine e la classe di appartenenza:

```
.
.
.
def get_network_boxes(net_ptr, w, h, thresh, hier, map_ptr, relative,
    num_ptr):
"""
net_ptr: pointer of network
```

```
w,h: image size
thresh: confidence threshold
return: pointer of array of DETECTION structure
"""
lib.get_network_boxes.argtypes = [
    ctypes.c_void_p,
    ctypes.c_int,
    ctypes.c_int,
    ctypes.c_float,
    ctypes.c_float,
    ctypes.POINTER(ctypes.c_int),
    ctypes.c_int,
    ctypes.POINTER(ctypes.c_int)]
lib.get_network_boxes.restype = ctypes.POINTER(DETECTION)
return lib.get_network_boxes(net_ptr, w, h, thresh, hier, map_ptr,
    relative, num_ptr)
.
.
.
```

Una volta caricato il modello e la rete neurale, è possibile visualizzare attraverso un canale video aperto tramite OpenCv gli oggetti rilevati e le relative informazioni.

4.3 Riconoscimento facciale

Per la parte del riconoscimento facciale si è dovuto procedere in due step: la prima parte un algoritmo veloce che riuscisse a riconoscere i volti in un'immagine e la seconda un algoritmo che riuscisse ad associare la corretta identità ai volti.

4.3.1 Face Detection

Per riconoscere i volti all'interno di un immagine, all'inizio si era pensato di utilizzare la libreria dlib per Python, tuttavia la libreria per essere performante necessitava di aver a disposizione memoria VRAM della GPU, cosa non possibile poichè la rete YOLOv3 consumava gia tutta la memoria a disposizione.
Inoltre si era cominciata la sperimentazione con il metodo di Viola-Jones[13] con un classificatore per i volti e i risultati ottenuti erano promettenti.
Tuttavia l'algoritmo non riusciva a riconoscere volti inclinati o parzialmente coperti, quindi si è optato per un metodo più sofisticato ed è stato quello

dell'utilizzo di una seconda rete neurale **Single Shot Detector** basata su ResNet-10 come struttura.
Il modello della rete è stato addestrato per moltissime iterazioni su un dataset molto ampio di volti umani (**FDDB**) ottendo in fase di validazione un AP (*Average Precision*) di 0.85 con IoU di 0.5 e un AR (*Average Recall*) di 0.482. Questo ha consentito alla rete di riconoscere volti anche da diverse angolazioni con moderata precisione.
Grazie alla libreria OpenCV, dalla versione 3.3 questa rete neurale e il relativo modello sono stati resi disponibili direttamente ed è utilizzabili con poche righe di codice:

```
import cv2
import os
current_path = os.path.dirname(os.path.realpath(__file__))
#the name of model file
caffe_model = "res10_300x300_ssd_iter_140000.caffemodel"
#the prototxt file that define the structure of the NN
prototxt = "deploy.prototxt"
#show faces only if confidence is higher of this value
threshold = 0.70
#load the neural network and the model from files
net = cv2.dnn.readNetFromCaffe(os.path.join(current_path,prototxt),
    os.path.join(current_path,caffe_model))
```

Una volta caricata la rete e il modello, bisogna convertire l'immagine da elaborare in un *blob*, ovvero un'immagine modificata a cui è stato applicato il *mean substraction*.
Il *mean substraction* è la sottrazione ad ogni canale di un'immagine (tre canali nel caso di un'immagine RGB) del valore medio del singolo canale su tutto il dataset.
Per esempio in un modello pre-addestrato su ImageNet i valori medi dei canali sono R = 103.93,G =116.77 e B = 123.68

```
#convert the image to blob for NET
blob = cv2.dnn.blobFromImage(cv2.resize(image, (300, 300)), 1.0,
(300, 300), (104.0, 177.0, 123.0))
net.setInput(blob)
#the net calculate all the bounding boxes of faces
detections = net.forward()
```

Convertita l'immagine in Blob e passata in input alla rete, come risultato si ottiene una lista di bounding boxes con relative posizioni all'interno dell'immagine, che verranno poi applicati allo stream video denotando i volti rispetto

alle altre componenti dell'immagine.

4.3.2 Face Recognition

Come visto nella sezione Face Detection, esistono librerie potenti e ottimizzate che riescono a eseguire riconoscimento facciale in maniera molto accurata, tuttavia le limitazioni hardware precedentemente menzionate sussistono anche per questo componente, perciò si è dovuto utilizzare un'altra tecnologia che non faceva uso di scheda grafica.
E' stato deciso quindi di utilizzare, dopo un'attenta analisi, un algoritmo noto alla letteratura ([14]) come **Local Binary Pattern Histogram** o (**LBPH**) che nonostante sia stato ideato nella prima versione nel 1994 e poi migliorato successivamente nel 1996, viene costantemente utilizzato e citato anche al giorno d'oggi[15] denotandone la sua efficienza.
La libreria OpenCV fornisce un'implementazione efficiente di questo algoritmo, facilmente importabile ed utilizzabile nel progetto.

4.3.2.1 Preparazione del Dataset e Training del modello

Essendo **LBPH** un algoritmo che esegue calcoli sui pixel, è importante la scelta del dataset poichè variazioni di luce, angolazione e sfocature possono risultare in un riconoscimento errato o non riconoscimento di un volto.
Il dataset che è stato usato per il progetto è un insieme di immagini che raffigurano il volto del tesista ripreso da diverse angolazioni e con illuminazione differente in modo da essere il più preciso possibile.
Attraverso un apposito tool scritto in Python, è possibile scattarsi delle foto (con webcam), le quali vengono convertite in scala di grigi e automaticamente il tool ritaglia da ogni immagine il volto (nel caso in cui venga rilevato) il quale verrà salvato in un file con estensione *.pgm* e aggiunto al dataset della persone indicata.
Un'ulteriore considerazione va fatto anche riguardo al numero di immagini utilizzate per il training: più il numero di immagini è elevato, maggiore è ovviamente anche la precisione, ma maggiore sarà anche il tempo di calcolo, rallentando quindi l'esecuzione del programma.
Si è valutato che con un numero di immagini non troppo elevato, circa 350, la dimensione del modello dopo la fase di training è di circa 25 MByte e che i tempi di esecuzione sono accettabili.
Qui di seguito vediamo come avviene il training del modello:

```
faces = []
labels = []
```

```
users = config.users
for (dirpath, dirnames, filenames) in os.walk(config.TRAINING_DIR):
    head, tail = os.path.split(dirpath)
    try:
        label = config.users.index(tail) + 1
        for filename in filenames:
            faces.append(prepare_image(os.path.join(dirpath,filename)))
            labels.append(label)
    except ValueError:
        pass

"""
[some print instruction for information during the process]
"""

#instance of the model
model = config.model(config.POSITIVE_THRESHOLD)
#train the model
model.train(np.asarray(faces), np.asarray(labels))
# Save model results in a file
model.write(config.TRAINING_FILE)
```

Una volta addestrato il modello le immagini e relative label, è stato testato con 100 immagini non appartenenti al dataset di train e si è ottenuto una confidenza media di 30, quando la soglia di rilevamento è 100, dimostrando che il modello è efficiente.
Una volta valutato il modello, lo possiamo importare creando un'istanza della seguente classe:

```
import cv2 # OpenCV Library
from .lib.face import face
from .lib.config import config
import time
import os
import signal
import sys

class FaceRecognition():
    """This class is used for face detection and recognition"""
    def __init__(self):
        #read the type model define in config.py
        self.model = config.model(config.POSITIVE_THRESHOLD)

        #load the trained model from the disk
```

```
        self.model.read(os.path.join(config.path_to_file,
        config.TRAINING_FILE))

    """
    [OTHER FUNCTIONS]
    """
```

Caricato il modello, è sufficiente richiamarlo come segue per verificare l'identità del volto:

```
#confidence lower, match stronger
label, confidence = self.model.predict(crop)
```

Il risultato di questa operazione è la precisione con cui è stata riconosciuto il volto e la relativa label.
Si riporta qui di seguito un'immagine di test che dimostra che il sistema funziona riuscendo a distinguere il tesista (Mattia) dal fratello nonostante la somiglianza.

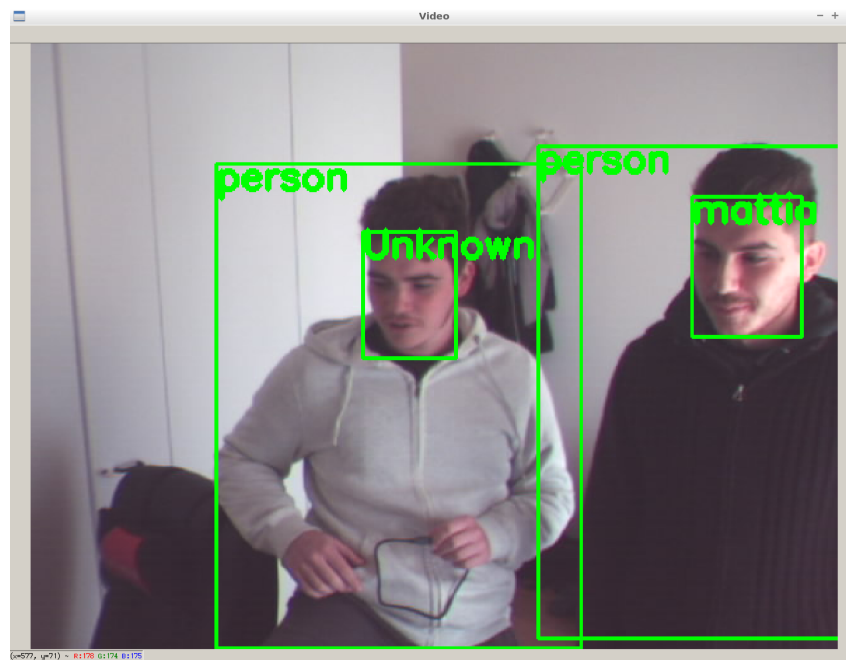

Figura 4.3: Il riconoscimento facciale distingue il soggetto del training dal fratello

Testato che anche l'ultimo componente funziona, si è proceduto a mettere assieme le tre parti per portare al risultato finale come esposto nella sezione successiva

4.4 Il sistema di sorveglianza

Come spiegato nell'introduzione, lo scopo di questa tesi è quello di prevedere un sistema di sorveglianza in ambienti indoor (per esempio un laboratorio) dove è di particolare importanza sapere se un determinato oggetto è ancora presente nell'ambiente e quali persone ci sono state al suo interno.
Posizionando una telecamera in una posizione strategica, è possibile avere una visione generale dell'ambiente permettendo di poter tener controllati sia gli oggetti che le persone.
Dal punto di vista del progetto, mettere assieme le tre componenti è stato relativamente semplice poichè i tre elementi sono stati pensati come moduli riutilizzabili, quindi facilmente importabili e configurabili secondo la politica di Python e della buona programmazione.
All'interno di un unico script in Python sono stati importati il il wrapper in Python di Darknet (darknet_libwrapper) e il modulo di riconoscimento facciale (Che include la fase di Face Detection e Recognition).

```
import sys
import ctypes
from darknet_libwrapper import *
from facerecognition.facerecognition import FaceRecognition
import cv2
import time
import numpy as np
import dictdiffer

metadata_file = 'cfg/coco.data'
config_file = 'cfg/yolov3.cfg'
weights_file = 'yolov3.weights'
#load the YOLO net
meta = get_metadata(metadata_file)
net = load_network(config_file, weights_file, 0)
set_batch_network(net, 1)
facerec = FaceRecognition()
```

Richiamando questi comandi viene inizializzata la rete YOLO e le sue strutture dati e caricati i modelli per il riconoscimento facciale.
Una volta caricate tutte le strutture dati necessarie, un ciclo while infinito cattura un'immagine dell'ambiente; l'immagine appena catturata viene pre-processata per essere compatibile con le varie componenti del sistema e successivamente è sottoposta ad elaborazione da parte degli stessi.

```
#cap is the webcamera
ret, img = cap.read()
if img is None:
    break
orig_img = img.copy()
#img have to be converted before sent it to YOLO
img = mat_to_image(img)
#_detector call function of Darknet wrapper
objects_detected = _detector(net,meta,img)
detected_faces = facerec.detect_bound_boxes_faces(orig_img)
```

Da un punto di vista prestazionale si è notato che effettuare il riconoscimento degli oggetti prima del riconoscimento dei volti garantiva sia migliori risultati che maggiori prestazioni. Una volta che il riconoscimento degli oggetti e dei volti ha avuto termine, si è proceduto ad una analisi delle informazioni acquisite. Il sistema per ogni persona cerca di associare e riconoscere il suo volto: nel caso in cui il sistema riconosca il soggetto entrato nel campo d'azione della telecamera, colora i bounding boxes della persona di verde segnalando che la persona è autorizzata.
Nel caso in cui invece il sistema non riesce a riconoscere un soggetto, il bounding boxes diventa di colore rosso e viene stampato a video e su un file di log locale l'eventuale intrusione di uno sconosciuto indicandone anche la data e l'ora.
Per il controllo degli oggetti invece si seleziona una lista di oggetti 'preferiti' tra tutti quelli nel dataset; in questo modo possiamo filtrare quali sono gli elementi che si, il sistema riesce a riconoscere, ma che non necessitano di essere tenuti sotto controllo.
Il sistema controlla costantemente se gli oggetti sono ancora presenti all'interno dell'ambiente e nel momento in cui un oggetto viene a mancare, viene segnalato con un messaggio sulla console e salvato l'evento nell'apposito file di log.
Inoltre il sistema è anche in grado di riaggiungere l'oggetto nel caso in cui venga rimesso nell'ambiente come definito nella porzione di codice sottostante.

```
if(len(local_objects) < len(objects)):
    #an object is removed
    removed_objects = list(set(objects) - (set(local_objects)))
    for obj in removed_objects:
        objects.remove(obj)
        print(f'Object {obj} removed at {datetime.datetime.now()}')
        log_file.write(f'Object {obj} removed at
            {datetime.datetime.now()}\n')
else:
    #an object is added
```

```
added_objects =
    list(set(local_objects).symmetric_difference(set(objects)))
for obj in added_objects:
    objects.append(obj)
    print(f'Object {obj} added at {datetime.datetime.now()}')
    log_file.write(f'Object {obj} added at
        {datetime.datetime.now()}\n')
```

Capitolo 5

Conclusioni e sviluppi futuri

Il sistema realizzato funziona, permettendo quindi di riconoscere oggetti e persone in un ambiente chiuso e la loro presenza, tuttavia le problematiche tecniche (webcam datata con risoluzione massima 640x480, specifiche hardware del calcolatore limitate) hanno portato a un risultato nettamente migliorabile soprattutto dal punto di vista delle prestazioni poichè non sono mai stati superati i 10 FPS.
Nonostante i buoni risultati ottenuti con l'attuale sistema, una delle migliorie apportabili al sistema è la sostituzione dell'algoritmo di riconoscimento facciale in favore della nota libreria in Python *face_recognition* che come riportano diversi siti del settore risulta essere al momento la più precisa e performante.
Continuare lo sviluppo di questo progetto con hardware più performante potrebbe portare al miglioramento dei risultati ottenuti sia in termini di efficienza che di precisione poichè potrebbe essere introdotta all'interno del sistema un'ulteriore Rete Neurale in grado di riconoscere le azioni delle persone nell'ambiente e di conseguenza capire se una persona ha preso un determinato oggetto oppure no.
Inoltre uno sviluppo futuro potrebbe essere l'utilizzo di una telecamera a infrarossi per la visione notturna in modo da poter controllare l'ambiente anche in assenza di illuminazione.

Ringraziamenti

Il primo ringraziamento va al relatore di questo lavoro, il Prof. Gianluca Moro, che ha reso possibile tutto ciò e ha acceso il mio personale interesse nei confronti di questa splendida disciplina.

Grazie anche alla mia famiglia, ai miei amici, ai miei colleghi e a chiunque ha condiviso con me questo tortuoso ma bellissimo percorso.

Bibliografia

[1] F. Rosenblatt., *The Perceptron: A Probabilistic Model For Information Storage And Organization*, Cornell Aeronautical Laboratory.

[2] David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams *Learning representations by back-propagating errors*, 1986.

[3] Alex Krizhevsky,Ilya Sutskever,Geoffrey E. Hinton *ImageNet Classification with Deep Convolutional Neural Networks*, 2012

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich *Going Deeper with Convolutions*, arXiv,17 Sep 2014

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun *Deep Residual Learning for Image Recognition*, arXiv,10 Dec 2015

[6] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*, arXiv:1506.02640, 2015

[7] Ross Girshick. *Fast R-CNN*, arXiv:1504.08083, 2015

[8] Joseph Redmon, Ali Farhadi. *YOLO9000: Better, Faster, Stronger* arXiv:1612.08242,2016.

[9] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* 2014.

[10] Joseph Redmon *Darknet: Open Source Neural Networks in C* 2013-2016

[11] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, Piotr Dollár *Focal Loss for Dense Object Detection* arXiv:1708.02002v2,2018.

[12] Zeming Li, Chao Peng, Gang Yu, Xiangyu Zhang, Yangdong Deng, Jian Sun *Light-Head R-CNN: In Defense of Two-Stage Object Detector* arXiv:1708.02002v2,2018.

[13] Paul Viola, Michael Jones *Rapid Object Detection using a Boosted Cascade of Simple Features* 2001

[14] Ahonen, T., Hadid, A., and Pietikainen, M. *Face Recognition with Local Binary Patterns* Computer Vision - ECCV 2004 (2004), 469–481.

[15] Aftab Ahmed,Jiandong Guo,Fayaz Ali,Farah Deeba *LBPH Based Improved Face Recognition At Low Resolution* 2018 International Conference on Artificial Intelligence and Big Data (ICAIBD), At Chengdu, China