

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Corso di laurea in Ingegneria Informatica

TESI DI LAUREA

in

Calcolatori Elettronici T

**CLASSIFICAZIONE DI IMMAGINI
MEDIANTE IL FRAMEWORK TORCH**

CANDIDATO

Ali Alessio Salman

RELATORE:

Prof. Stefano Mattoccia

CORRELATORE

Dott. Matteo Poggi

Anno Accademico 2013/2014

Sessione II

Indice

Indice.....	3
1. INTRODUZIONE.....	7
2. VISIONE ARTIFICIALE	9
2.1 Premessa.....	9
2.2.1 Pattern Recognition.....	10
2.2 La classificazione.....	11
2.2.1 Definizione	11
2.2.2 Estrazione delle caratteristiche	12
2.2.2.1 Pre-processing.....	13
2.2.2.2 Riconoscimento dei contorni.....	15
2.2.2.3 Segmentazione	17
2.2.2.4 Texture e colore	18
2.3 Esempi di classificatori	19
2.3.1 Classificatore Bayesiano	19
2.3.2 Support vector machines.....	20
2.3.3 Logistic Regression.....	21
3. MACHINE LEARNING.....	23
3.1 Introduzione	23
3.2 Paradigmi di apprendimento.....	24
3.3 Reti Neurali.....	27
3.3.1 Fondamenti biologici e matematici	27
3.3.2 Analisi del funzionamento.....	27
3.3.3 Tipologia di reti neurali artificiali trattate.....	30

3.3.4 Pregi e difetti.....	31
3.4 Multilayer Perceptron.....	32
3.4.1 Struttura	32
3.4.1.1 Strati Nascosti.....	33
3.4.2 Apprendimento mediante Back-Propagation.....	34
3.5 Deep Learning.....	35
4. CONVOLUTIONAL NEURAL NETWORKS	37
4.1 Storia	37
4.2 Architettura	38
4.2.1 Panoramica Generale	38
4.2.2 Convoluzione.....	39
4.2.3 Subsampling.....	43
4.3 Pregi e peculiarità	45
4.4 Utilizzi e prestazioni	47
4.5 Confronto con l'uomo	49
5. IL FRAMEWORK TORCH.....	51
5.1 Introduzione	51
5.1.1 Supporto CUDA.....	54
5.2 Installazione.....	54
5.3 Utilizzo	56
5.3.1 Premessa	56
5.3.2 Dataset.....	56
5.3.2.1 Il package Image.....	59
5.3.3 Definizione di una CNN	59

5.3.3.1 Note importanti	61
5.3.4 Loss function	61
5.3.5 Training della rete.....	63
5.3.6 Testing della rete	64
5.4 Esempi applicativi.....	66
5.4.1 MNIST	66
5.4.2 CIFAR10.....	67
5.5 Casi di studio	73
6. RISULTATI SPERIMENTALI E CONFRONTO CON IL FRAMEWORK CAFFE	75
6.1 Premessa.....	75
6.2 MNIST.....	76
6.3 CIFAR10	77
7. CONCLUSIONI E SVILUPPI FUTURI.....	79
<i>Ringraziamenti.....</i>	80
Riferimenti.....	81

1. INTRODUZIONE

Nella maggioranza degli organismi viventi il riconoscimento delle configurazioni assunte dall’ambiente esterno è svolto da complesse organizzazioni di cellule nervose, le quali si occupano anche della memorizzazione e delle reazioni agli stimoli provocati dallo stesso. Il cervello umano rappresenta probabilmente il più mirabile frutto dell’evoluzione per le sue capacità di elaborare informazioni. Al fine di compiere tali operazioni, le reti biologiche si servono di un numero imponente di semplici elementi computazionali – i neuroni – fortemente interconnessi, in modo da variare la loro configurazione in risposta agli stimoli esterni: in questo senso si può parlare di apprendimento ed i modelli artificiali cercano spesso di riprodurre questo tratto biologico.

Una rete neurale artificiale riceve segnali esterni su uno strato di neuroni, unità di elaborazione, i quali sono poi connessi ad altri neuroni internamente secondo una struttura che può avere un numero di livelli dipendente dalla tipologia della rete. La comunicazione tra queste unità di elaborazione è altamente ispirata dallo studio biologico del nostro cervello. Di tali reti si parlerà ampiamente nei paragrafi successivi.

In particolare la tesi si è concentrata sulla sperimentazione di algoritmi per un’efficiente classificazione e riconoscimento di immagini appartenenti a un *dataset*, mediante un approccio allo stato dell’arte, basato su *Convolutional Neural Networks (CNNs)* [1].

Per sperimentare il suddetto approccio si è fatto uso di Torch7 [2], un framework *open source* con ampio supporto al *Deep Learning* [3]. Torch è stato realizzato inizialmente da Ronan Collobert all’IDIAP Research Institute in Svizzera ed è tutt’ora in fase di sviluppo sia da parte dell’autore che da parte della New York University (NYU), i NEC Laboratories America, Clement Farabet, Koray Kavukcuoglu, Soumith Chintala, e la comunità di utenti che ne fanno utilizzo come software libero.

La prima parte della tesi introduce il tema della visione artificiale, gli scopi e il problema della classificazione delle immagini e i possibili metodi utilizzabili. Si analizzeranno in seguito gli algoritmi di *Machine Learning* (apprendimento automatico) e il sottoinsieme più specifico del *Deep Learning*.

Particolare attenzione è stata dedicata alle reti neurali, alla loro architettura e funzionamento e soprattutto al loro utilizzo. Nel capitolo 5 sono studiate e spiegate in dettaglio le *Convolutional Neural Networks* ed è presentato poi nel capitolo 6 l'utilizzo del framework Torch, basato su questo approccio. La parte fondamentale del mio lavoro è stata quella di utilizzare tale framework adoperando come ausilio la documentazione e gli esempi applicativi reperibili sul sito del progetto [24]

Una volta apprese le metodologie necessarie per sviluppare progetti con Torch, si è sfruttato tale framework per il riconoscimento di immagini appartenenti ad un dataset estratto tramite una telecamera 3D con elaborazione su FPGA sviluppata presso il DISI al fine di categorizzare oggetti inquadrati dal sistema di visione nel contesto di un sistema di ausilio per non vedenti.

La parte finale è dedicata ad un confronto con Caffe [3], un altro framework per il deep learning analizzato da Elisa Rimondi nel suo elaborato “Classificazione di immagini mediante il framework Caffe” svolto in parallelo.

Completano il lavoro le conclusioni e una riflessioni sulle possibili estensioni future.

2. VISIONE ARTIFICIALE

2.1 Premessa

Gli esseri umani (e la maggior parte degli esseri viventi) sono una specie altamente dipendente dal senso della visione, e usano gli occhi costantemente per valutare il mondo circostante. Con occhi rivolti in avanti come gli altri primati, usiamo la visione per percepire quei molti aspetti dell'ambiente che sono lontani dal nostro corpo.

La luce è una forma di energia elettromagnetica che entra nei nostri occhi e agisce sui fotorecettori posti sulla retina. Questo dà l'avvio a processi attraverso i quali sono generati *impulsi neurali* che attraversano i percorsi e le reti di quelle parti del cervello dedicate alla visione. Esistono percorsi separati che raggiungono il ponte e la corteccia cerebrale per mediare diverse funzioni rilevare e rappresentare il movimento, le forme, i colori ed altri caratteri distintivi del mondo visibile. Nella corteccia, i neuroni in un gran numero di aree visive distinte, sono specializzati nel compiere diversi tipi di decisioni visive, in base al contenuto visivo che percepiamo.

La corteccia visiva è costituita da un certo numero di aree, ciascuna delle quali è dedicata ad uno degli aspetti del mondo visibile quali la forma, il colore, il movimento, la distanza, ecc. Le sue cellule sono organizzate in colonne. È stato importante scoprire che tutti i neuroni che compongono ogni singola colonna rispondono maggiormente a linee o margini con uno specifico orientamento, mentre la colonna adiacente risponde meglio a linee o margini con un orientamento leggermente diverso dal precedente, e così via su tutta la superficie della corteccia visiva. Questa scoperta è stata sfruttata nell'implementazione dei filtri di Gabor [4].

Questo significa che le cellule della corteccia visiva hanno un'organizzazione intrinseca per l'interpretazione del mondo, ma non è un'organizzazione che non cambia mai. La varietà di direzioni alla quale una singola cellula può essere reattiva è modificata dall'esperienza attraverso i segnali che provengono dall'occhio destro o sinistro. Come per tutti i sistemi sensoriali, la corteccia visiva manifesta anch'essa quella capacità di cui siamo dotati, chiamata plasticità. Davide Hubel e Tornsten Wiesel ricevettero il Nobel per la Medicina nel 1981 per la scoperta di queste affascinanti caratteristiche della cellula della corteccia visiva.

Un concetto importante riguardo la capacità reattiva delle cellule è quello del campo recettivo (in inglese, *receptive field*) : la porzione di retina sulla quale le cellule rispondono ad un tipo di immagine piuttosto che ad un altro. Tale concetto sarà utilizzato nelle convolutional neural network.

L'uomo elabora le immagini in maniera qualitativa, mediante un processo decisionale ad alto livello, che utilizza le informazioni spaziali ricevute tramite l'occhio e crea astrazioni di queste internamente. Tale elaborazione qualitativa è però limitata nella distinzione dei livelli di grigio e dei toni di colore. Le macchine dal loro canto – attraverso un'elaborazione quantitativa – riescono a processare in maniera utile tutte le scale di grigio, ma non hanno la capacità di cogliere le informazioni semantiche in essa. Tuttavia, utilizzando questa capacità oggettiva delle macchine di elaborare le immagini, si possono progettare diversi metodi efficienti per il riconoscimento e la classificazione di oggetti.

La visione artificiale è l'insieme dei processi che mirano a creare un modello approssimato del mondo reale partendo da immagini (tipicamente bidimensionali ma anche di diversa natura come immagini *depth*). Lo scopo principale di questo ramo dell'intelligenza artificiale è proprio quello di riprodurre la vista umana, in un senso collegato soprattutto ad acquisire, come detto prima, la capacità dell'interpretazione dell'informazione contenuta in un'immagine per potere poi prendere una decisione oculata.

2.2.1 Pattern Recognition

Gli esseri umani hanno la capacità di riconoscere e classificare oggetti visuali che rispettano certe caratteristiche di modelli teorici, detti anche *pattern*, attribuendogli un nome e un'identità. La tesi si inquadra appunto in un settore, chiamato *pattern recognition*, che studia e sviluppa metodi per il riconoscimento di pattern specifici nei dati, acquisendo la capacità di classificarli. Molto spesso questi metodi fanno utilizzo di algoritmi di apprendimento supervisionato (lo si vedrà in dettaglio nel capitolo 3). Attraverso l'analisi dei pattern che differenziano un oggetto dall'altro, questi metodi sono in grado di classificare le immagini, rendendo quindi possibile la realizzazione di sistemi che prendano decisioni in modo intelligente rispetto a quello che “vedono”.

2.2 La classificazione

2.2.1 Definizione

La classificazione è il problema di identificare a quale set di categorie appartiene un dato in ingresso, usualmente mediante un allenamento su di un *training set* (insieme di esempi). Ognuno degli esempi appartenenti al training set è preventivamente “categorizzato” tramite un’etichetta, in modo da insegnare al sistema a riconoscere i pattern specifici per quella determinata categoria. Un esempio può essere quello di discriminare una mail di spam da una mail normale oppure quello di fare una diagnosi di un paziente sulla base di alcuni fattori di riferimento (sesso, pressione sanguigna, sintomi, ecc.).

Esistono anche metodi di classificazione basate su un apprendimento *non supervisionato* note con il nome di “clustering”, da “cluster” cioè gruppo, nel senso di raggruppare, ovvero categorizzare in base a pattern creati direttamente a partire dai dati senza istruzioni preventive. Una più curata spiegazione di entrambi si vedrà nel capitolo 3.

I sistemi di classificazione debbono, come detto, riuscire a classificare dati in ingresso secondo dei pattern, ma prima di tutto bisogna avere sistemi capaci di *estrarre* le informazioni peculiari direttamente dall’immagine, attraverso diversi passaggi. L’insieme di questi passaggi prende il nome di *feature extraction*, estrazione delle caratteristiche.

Raggruppando le caratteristiche peculiari delle immagini si viene a creare un cosiddetto “spazio delle caratteristiche” [5]. Lo spazio delle caratteristiche è uno spazio n-dimensionale dove n è il numero delle bande spettrali di ogni pixel, ovvero il numero di canali in cui è conservata l’informazione di ognuno di essi.

2.2.2 Estrazione delle caratteristiche

L'estrazione delle caratteristiche è una maniera particolare per ridurre la dimensionalità dei dati e quindi le risorse necessarie per l'elaborazione degli stessi. Quando i dati sono troppi c'è il rischio di ridondanza delle informazioni; pertanto i dati sono elaborati e convertiti in una rappresentazione ridotta, che può essere vettoriale o matriciale (si parla di feature vector o feature map).

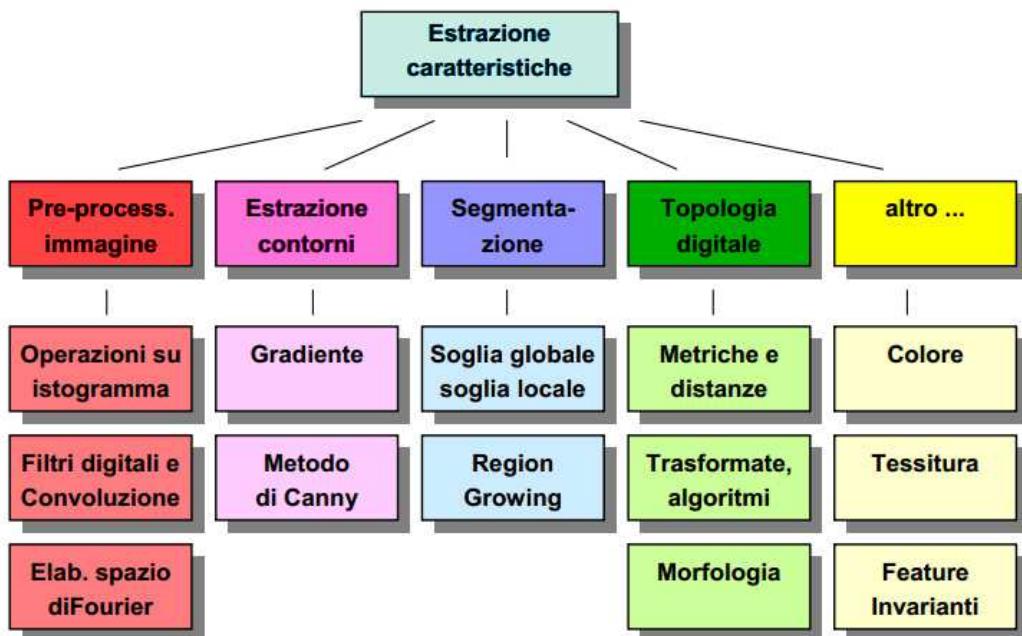


Figura 2.1 : I vari step dell'estrazione delle caratteristiche

Tale riduzione di dimensionalità comporta anche una diminuzione delle variabili in gioco, che migliora la gestione della memoria e semplifica di molto l'elaborazione, in particolar modo quando queste variabili non sono completamente automatizzate ma devono essere calibrate anche tramite un'analisi manuale.

Tale riduzione può però essere deleteria se non si riesce a mantenere un buon grado di accuratezza delle informazioni dei dati in ingresso, per poterle elaborare – nel nostro caso

classificare – correttamente. Per questo sono stati creati metodi automatici e affidabili per eseguire questo compito.

L'estrazione delle caratteristiche è divisa in più passi:

- pre-processing dell'immagine
- riconoscimento dei contorni
- segmentazione
- texture e colore

2.2.2.1 Pre-processing

La prima fase è quella del preprocessing, che può essere molto importante, in quanto permette un notevole miglioramento delle informazioni disponibili mediante la manipolazione delle immagini per l'estrazione delle caratteristiche.

Tipicamente si lavora sull'istogramma dell'immagine, che traccia il numero di pixel per ogni valore tonale. Si procede pertanto a una *equalizzazione* (o normalizzazione), ossia una calibrazione del contrasto dell'immagine cercando di far ottenere lo stesso risalto a tutte le parti dell'immagine, evitando di avere parti troppo scure e difficilmente riconoscibili.

Questo metodo di solito incrementa il contrasto globale di molte immagini, specialmente quando i dati usabili dell'immagine sono rappresentati da valori di contrasto molto vicini.



Figura 2.2: Equalizzazione di una immagine rappresentata con livelli di grigio

Attraverso questo adattamento, le intensità possono essere meglio distribuite sull'istogramma. Questo permette alle aree a basso contrasto locale di ottenere un più alto

contrasto. Tale effetto si ottiene distribuendo maggiormente la dinamica dei livelli di grigio o dei colori.

Dopodiché si possono usare diversi strumenti a seconda del risultato che si vuole ottenere.

Per esempio, si utilizzano spesso filtri di *smoothing* per ridurre il rumore e le variazioni di luminosità. Oltre che allo smoothing, ci si serve di filtri di *sharpening* grazie ai quali si evidenziano i dettagli dell'immagine e i suoi contorni: l'effetto desiderato si ottiene sommando la risposta del filtro all'immagine originale.

Si possono usare metodi di *pooling* ovvero metodi per ottenere da un pool (un insieme, in questo caso una matrice) di pixel, un solo pixel che sia correlato in maniera matematica al suo pool. Si fa scorrere sull'immagine una matrice di pooling di dimensione arbitraria: facendo un pooling di $N \times N$ vuol dire che si andrà ad estrarre 1 pixel ogni N^2 .

Vedremo in seguito diversi metodi per scegliere il pixel.

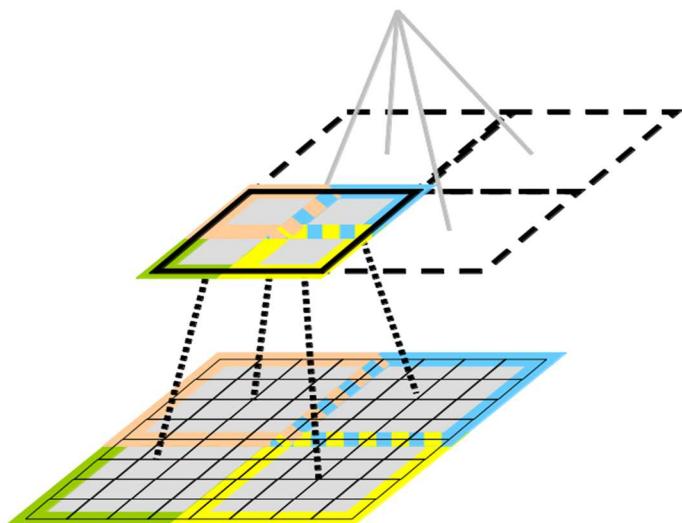


Figura 2.3 : Esempio di max pooling

2.2.2.2 Riconoscimento dei contorni

Il riconoscimento dei contorni (*edge detection*) è utilizzato allo scopo di marcare i punti dell'immagine in cui l'intensità luminosa cambia bruscamente, o più formalmente, si ha una discontinuità. Tali discontinuità sono di solito indicative di un cambiamento importante del mondo fisico di cui le immagini sono la rappresentazione. Suddetti cambiamenti possono, ad esempio, essere dovuti a: discontinuità della profondità, modifica delle proprietà dei materiali, variazioni di illuminazione dell'ambiente circostante.

Nel caso ideale, applicare un edge detector può condurre ad avere un set di curve connesse che possono indicare il contorno di un oggetto oppure una discontinuità sulla superficie dello stesso, che può essere causata da un cambiamento di orientamento. L'immagine risultante, quindi, contiene sicuramente molte meno informazioni di quella su cui è applicato il filtro. Sono cioè eliminate molte delle informazioni meno rilevanti, cercando comunque di mantenere le proprietà strutturali e geometriche dell'immagine. Se questo avviene con successo, le successive elaborazioni saranno notevolmente semplificate; non è tuttavia sempre facile da ottenere con immagini del mondo reale, anche di moderata complessità.

La maggior parte dei metodi per riconoscere i contorni si basa sulla ricerca delle massime variazioni di intensità tramite il calcolo del gradiente, ovvero le derivate parziali nelle due direzioni delle immagini.

Uno dei metodi più efficaci e utilizzati è il filtro di Canny. Utilizza un metodo di calcolo multi-stadio per individuare i contorni (edge) delle immagini. Si compone di diverse fasi di elaborazione.

- Utilizza prima 4 filtri per evidenziare i contorni orizzontali, verticali e diagonali e poi procede con la ricerca dei massimi locali del gradiente, come descritto (in maniera semplificata) prima.
- Esegue una soppressione di tutti i punti che non sono massimi locali.
- Infine esegue una fase chiamata sogliatura con isteresi, dove si definiscono 2 soglie, una bassa e una alta che sono confrontate col gradiente di ogni punto dell'immagine. Se il gradiente è:

- a) inferiore alla soglia bassa il punto è scartato
- b) superiore alla soglia alta, il punto viene accettato come parte di un contorno
- c) compreso tra le soglie, il punto è accettato solamente se contiguo ad un punto accettato in precedenza

Nella fig. 2.4 vi è un esempio dell'effetto di vari edge-detector, compreso l'algoritmo di Canny.

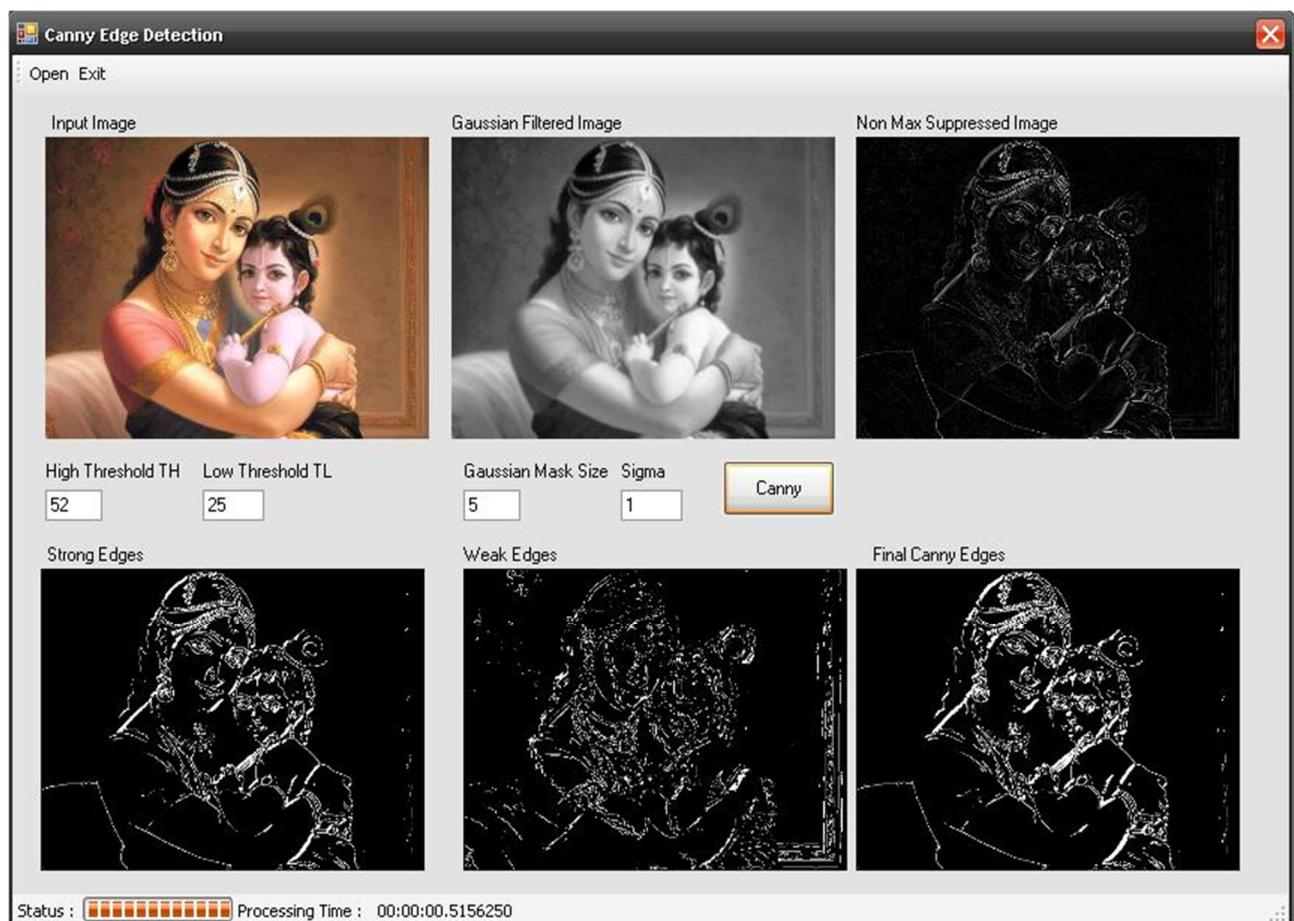


Figura 2.4 : Immagine originale e effetti di diversi edge-detector

2.2.2.3 Segmentazione

La segmentazione si pone come obiettivo quello di classificare pixel dell'immagine che abbiano caratteristiche simili tra loro e che siano connessi. Pertanto, ciascun pixel in una regione è simile agli altri della stessa regione per una qualche proprietà o caratteristica (colore, intensità o *texture*). L'insieme di queste regioni deve coprire l'intera immagine originale e – come possiamo vedere nella figura d'esempio – nessuna regione deve essere sovrapposta con nessun'altra. Anche questo processo produce un'immagine più semplice da elaborare e analizzare.



Figura 2.5: Esempi di segmentazione

Come possiamo vedere dall'esempio e da quanto detto prima l'immagine segmentata deve rispettare le seguenti proprietà:

- nessun pixel deve essere condiviso fra due regioni
- tutti i pixel appartenenti a una regione sono connessi tra loro
- tutti i pixel dell'immagine sono assegnati ad almeno una regione della partizione
- tutte le regioni sono omogenee rispetto ad un criterio prefissato

Nel caso più semplice di oggetto singolo, quindi distinto da una soglia di intensità di pixel globale, il metodo consiste nell'annullare i pixel inferiori a tale soglia e impostare a 1 i rimanenti. Per fare ciò, si utilizza l'istogramma dell'immagine dove la presenza di picchi di intensità è causata da oggetti a luminosità diverse.

Quando però gli oggetti nell'immagine sono diversi, oppure quando lo sfondo presenta molteplici variazioni graduali di luminosità è opportuno ricorrere ad altri metodi. Uno di questi è il *region growing* (accrescimento della regione) che, partendo da un insieme di punti distinti che rappresentano regioni disgiunte dell'immagine, accresce queste regioni sino a che tutta l'immagine sia coperta. In questo caso è importante definire una maniera con la quale scegliere i pixel *seed* di partenza e una regola con la quale aggiungere pixel a ciascuna regione originata da un seed.

2.2.2.4 Texture e colore

Per la segmentazione, la localizzazione e il *matching* di pattern, esistono spazi colore diversi da RGB che possono essere utilizzati in maniera più adeguata per il raggruppamento spaziale di quei colori avvertiti dall'uomo come simili.

Un esempio è lo spazio YUV, che definisce un canale di luminanza (Y) e due canali di crominanza (U,V). Questo spazio colore è molto utilizzato per la codifica di immagini e video, perché permette un miglior controllo degli errori di trasmissione. È stato studiato per la televisione e concepito per emulare la visione umana. Vedremo che sarà usato anche in alcuni algoritmi implementati con Torch, poiché isolare il canale della luminanza certe volte risulta molto comodo, in quanto stessi oggetti possono avere colorazioni differenti a seconda della luce che li colpisce, quindi il colore non è, in generale, un buon metodo per classificarli.

Un altro spazio colore da menzionare è l'HSV (Hue Saturation Value), basato sulla percezione che si ha di un colore in termini di tinta, sfumatura e tono.

Ci sono comunque altri spazi colore, alcuni sono variazioni dei modelli appena descritti.

Talvolta le immagini presentano quella che viene chiamata texture. Le texture sono trame più o meno regolari caratterizzate da alcune primitive che ne descrivono la struttura. Ci sono diversi metodi per la discriminazione delle texture.

Un modo è quello di usare filtri particolarmente utili in questo compito, come i filtri di Gabor. Sono filtri lineari utilizzati per l'edge detection e, come detto nella sezione introduttiva, hanno una frequenza e una rappresentazione dell'orientamento degli edges simile a quella del sistema visivo umano. Nello spazio 2D un filtro di Gabor è un'onda sinusoidale attenuata progressivamente da una Gaussiana. Queste funzioni possono modellare in maniera ottimale le cellule semplici della corteccia visuale del cervello dei mammiferi.

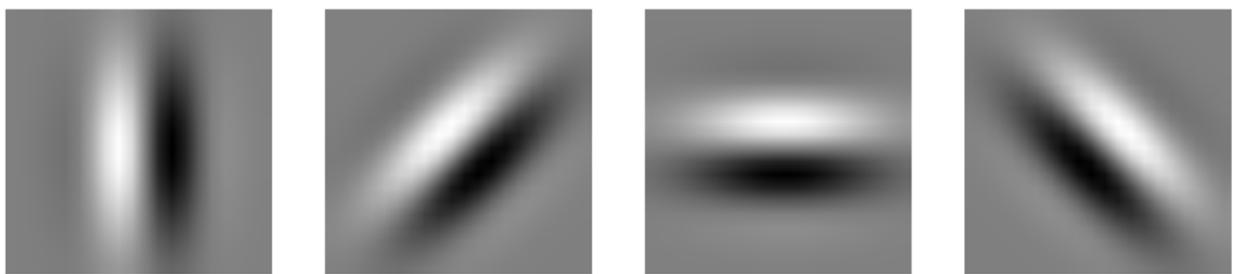


Fig. 2.6 : 4 filtri di Gabor, sensibili a diversi orientamenti

2.3 Esempi di classificatori

2.3.1 Classificatore Bayesiano

Un classificatore molto utilizzato si basa sul teorema di Bayes, che viene utilizzato per calcolare la probabilità di una causa che ha scatenato un evento già verificato.

Il classificatore bayesiano richiede quindi la conoscenza delle probabilità a priori e condizionali relative al problema, generalmente ricavabili tramite una stima. Se queste stime sono abbastanza affidabili, allora questo classificatore risulta efficace. Spesso viene chiamato classificatore bayesiano completo o *Belief Network*.

Facendo l'ipotesi che le features siano indipendenti fra loro possiamo vedere un esempio: sia dato il vettore $A=(A_1, A_2, \dots, A_n)$ che descrive il set di attributi e sia C la variabile di classe. Se C è legato in modo non deterministico ai valori assunti da A possiamo trattare le due variabili come variabili casuali e catturare le loro relazioni probabilistiche utilizzando la probabilità condizionata $P(C|A)$. Durante la fase di training si imparano i legami probabilistici $P(C|A)$ per ogni combinazione di valori assunti da A e C .

Conoscendo queste probabilità, si può classificare un esempio nuovo A, trovando la label di classe C che massimizza la probabilità a posteriori $P(C|A)$.

Nelle ipotesi dette sopra, il modello è realizzabile con maggiore facilità e la verifica dimostra che è anche efficace in molti problemi pratici, come per esempio il filtraggio anti-spam di cui si è parlato nella sezione 2.2.1.

2.3.2 Support vector machines

Le *support vector machines* sono un insieme di metodi di apprendimento supervisionato ideati nel 1993-95 da Vladimir Vapnik e il suo team presso i Bell AT&T Laboratories [25]. Appartengono alla famiglia dei classificatori lineari generalizzati e sono anche noti come classificatori a massimo margine, poiché allo stesso tempo minimizzano l'errore empirico di classificazione e massimizzano il margine geometrico. Le SVM sono un metodo particolarmente efficace per la classificazione di pattern, e rappresentano un'alternativa all'addestramento delle reti neurali. Una rete neurale a un solo livello non è in grado di classificare dati non linearmente separabili; mentre le reti neurali a più livelli possono farlo ma richiedono un addestramento più complesso e richiedono di risolvere un problema di ottimizzazione non convesso, il che non è sempre fattibile, soprattutto se il numero di variabili in gioco è molto alto.

Al contrario, le SVM sono un algoritmo efficace e in grado di rappresentare funzioni non lineari, anche complesse. Esse costruiscono un iperpiano, o un set di iperpiani, in uno spazio dimensionale anche elevato, cercando di dividere le classi del problema in modo geometrico, trovando i cosiddetti *classification boundaries*.

Per capire meglio, immaginiamo di dover classificare due classi in uno spazio bidimensionale. Queste due classi avranno elementi sparsi nel piano, e nel caso si possa trovare una maniera di separarli con una retta le SVM sapranno trovarla, e quella retta sarà quella ottimale. Nel caso di più dimensioni questa separazione avviene tramite un iperpiano. Questa divisione geometrica divide lo spazio in due parti: ogni successivo dato in ingresso sarà classificato dal sistema come appartenente ad uno dei due, in base a quali delle due regioni contiene il punto dello spazio individuato da questo elemento.

La scelta è fatta in maniera ottimale tramite un criterio basato su margine funzionale, ovvero si sceglie (nel caso in 2D) la retta che ha la distanza più grande da ogni punto più vicino di ogni classe. Un esempio è nella figura 2.7.

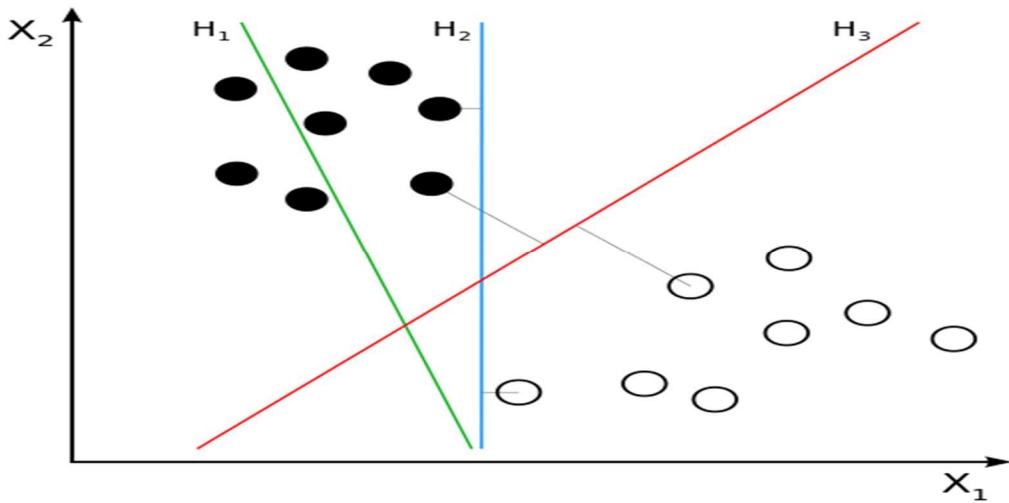


Figura 2.7: H1 non separa le classi; H2 le separa ma con un margine basso; H3 le separa con un margine migliore.

Può accadere che la separazione completa delle due classi utilizzando un iperpiano non sia possibile. Qualunque iperpiano si scelga ci saranno alcuni punti di una classe che si trovano dalla parte dello spazio associata all'altra classe. In questo caso si introduce un costo associato agli errori di classificazione nell'ottimizzazione, modificando di fatto l'algoritmo.

Yichuan Tang dimostra[11] che, utilizzando una SVM alla fine di una *convolutional neural network* al posto di una normale rete neurale, si abbiano discreti miglioramenti di performance su tutti i più popolari dataset.

2.3.3 Logistic Regression

La *logistic regression* è un classificatore probabilistico lineare. Quando le dimensioni del problema sono maggiori di due allora si parla di *multinomial logistic regression*.

La classificazione è eseguita proiettando i dati (punti nello spazio delle features) su un set di iperpiani, la distanza dai quali riflette la probabilità di appartenenza ad una data classe.

3. MACHINE LEARNING

3.1 Introduzione

L'espressione "Intelligenza Artificiale" fu coniata nel 1956 dal matematico statunitense John McCarthy, durante uno storico seminario disciplinare svoltosi nel New Hampshire: con questo termine si intende generalmente l'abilità di una macchina di svolgere compiti e ragionamenti tipici di un essere umano.

Il *machine learning* (noto anche come apprendimento automatico) è una disciplina scientifica concernente lo sviluppo di sistemi e algoritmi che possono imparare dall'analisi dei dati. È considerato un campo specializzato dell'informatica e la statistica, ma è anche una delle aree fondamentali dell'intelligenza artificiale, poiché si occupa di sintetizzare nuova conoscenza partendo dall'osservazione di dati anche complessi, dalle loro caratteristiche di interesse e le loro peculiarità, per ottenere sistemi capaci di prendere decisioni autonome.

Una definizione formale che include qualsiasi programma per computer capace di migliorare le sue prestazioni con l'esperienza è stata data da Tom M.Mitchell [4]: “*un programma apprende da una certa esperienza E se: nel rispetto di una classe di compiti T, con una misura di prestazione P, la prestazione P misurata nello svolgere il compito T è migliorata dall'esperienza E'*”.

I dati che si analizzano possono essere enormi (si pensi ad esempio all'insieme delle ricerche su Google) e tutti i possibili loro comportamenti sono troppi da poter essere coperti da un insieme di esempi osservati. Nasce quindi la difficile esigenza di creare sistemi capaci di comprendere grosse moli di dati ed evolversi di conseguenza, in grado di produrre un rendimento utile per i continui nuovi casi in ingresso adattandosi e migliorando le proprie capacità nello svolgere un particolare compito. Il machine learning è quindi impiegato in tutti quei campi in cui progettare un esplicitamente un algoritmo con delle regole fisse è infattibile. È un settore fortemente multidisciplinare che include ad esempio: il riconoscimento ottimale dei caratteri (OCR), motori di ricerca, applicazioni biomediche per il rilevamento di cellule malate, e molte applicazioni in ambito visione artificiale.

3.2 Paradigmi di apprendimento

Ci sono diversi paradigmi di apprendimento che si dividono principalmente in cinque tipologie, di seguito elencate:

- **Apprendimento supervisionato:** con questo termine s'intende l'allenamento di un sistema tramite una serie di esempi ideali; l'insieme di questi esempi è chiamato training set. Il sistema impara quindi ad approssimare una funzione non nota a priori a partire da una serie di coppie ingresso-uscita: per ogni input in ingresso gli si comunica l'output desiderato (si dice che gli esempi sono *labeled*, ovvero etichettati). L'apprendimento consiste nella capacità del sistema – tramite la funzione generata con l'allenamento – di generalizzare a nuovi esempi: avendo in ingresso dati non noti deve poter predire in modo corretto l'output desiderato.

Matematicamente parlando, questi esempi non noti sono punti del dominio che non fanno parte dell'insieme degli esempi di training.

Siano x_i, y_i rispettivamente gli input e gli output della rete. Esistono diversi algoritmi di apprendimento supervisionato ma *tutti condividono una caratteristica: l'allenamento viene eseguito mediante la minimizzazione di una funzione di costo (la cosiddetta loss function)* la quale rappresenta l'errore dell'output y_i fornito dalla rete rispetto all'output desiderato \tilde{y} .

Una funzione di costo spesso utilizzata è lo scarto quadratico

$$\frac{1}{2} \sum_i (y_i - \tilde{y})^2 \quad (3.1)$$

La scelta della funzione da minimizzare è importante poiché essa sottintende il principio che sta alla base dell'apprendimento. Utilizzando una funzione come quella dello scarto quadratico, che computa sull'insieme di *tutti* gli esempi di training significa supporre che minimizzando l'errore su tutti gli esempi si minimizzi l'errore anche sugli esempi non noti. Nella pratica è buona norma *non* prendere per buono questo principio, soprattutto se il training set non è molto vasto. Questo perché,

minimizzando quella tipologia di funzione, la rete tende a legarsi in maniera troppo specifica agli esempi del training set e quando l'input varia, magari anche di poco, si corre il rischio di avere una predizione errata. In letteratura questo problema è noto come *overfitting* [5]. Una possibile soluzione a questo dilemma è dividere il training set in due parti uno per l'allenamento (*training*) e uno per una verifica (*validation*). Si usa quello più piccolo per controllare l'efficienza dell'addestramento, valutando la loss function su di esso. Siccome questo può non bastare, spesso si ricorre a un ultimo insieme di dati, detto test set. Se l'errore su questo insieme è simile a quello del training allora non è avvenuto overfitting.

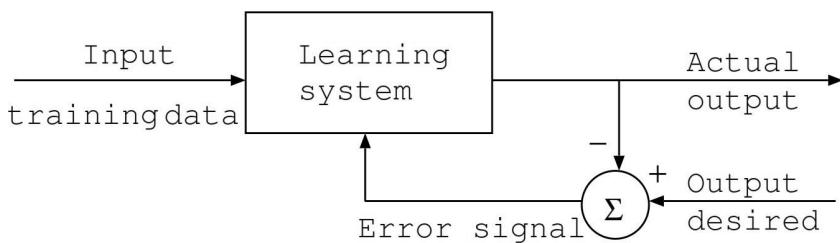


Figura 3.1 Schema semplificato di un apprendimento supervisionato [27]

Questo è il tipo di apprendimento che abbiamo scelto di utilizzare per il nostro problema. Nella prossima sezione saranno spiegati in dettaglio i vari passi dell'apprendimento, mentre degli altri paradigmi verrà fornita solo una breve panoramica.

- Apprendimento non supervisionato: il problema diventa quello di trovare strutture nascoste in dati non preclassificati per cui non è possibile valutare una soluzione (e quindi una funzione di costo da minimizzare). Questo tipo di approccio ha però il vantaggio di non richiedere grandi quantità di esempi categorizzati, siccome le categorie le crea il sistema stesso autonomamente: analizzando i vari dati in ingresso esso li dividerà in gruppi, cluster, che hanno caratteristiche comuni e che si differenziano gli uni dagli altri. Potrà ad esempio capire autonomamente che i cavalli sono diversi dalle porte le quali sono diversi dalle navi, inserendoli in 3 gruppi separati.

- Apprendimento con rinforzo: questa tecnica punta a ottenere sistemi capaci di cambiare nell'ambiente in cui eseguono secondo una ricompensa o una penalità che prende il nome di *rinforzo*.

Per fare ciò si deve disporre di 3 meccanismi:

- a) un meccanismo logico A che deve essere una sorta di classificatore, cioè essere in grado di scegliere un output in base ad un determinato input.
- b) un meccanismo logico B in grado di valutare l'efficacia dell'output del meccanismo A in base ad un preciso parametro di riferimento. Questo si occupa di mandare in output una ricompensa/penalità secondo le scelte fatte da A.
- c) un meccanismo logico C che deve modellare una funzione matematica in base agli output di B, ossia le ricompense/penalità.

Questa funzione matematica definisce il comportamento di A cercando di avere il maggior numero di ricompense possibili.

- Esperienza con apprendimento continuo: sono algoritmi basati sul principio di disporre di un meccanismo semplice in grado di valutare le proprie scelte e quindi premiare o punire l'algoritmo a seconda del risultato. Sono in grado di adattarsi anche a modifiche sostanziali dell'ambiente: un esempio sono i programmi di riconoscimento del parlato o i programmi di OCR che con l'utilizzo migliorano le loro prestazioni.
- Esperienza con addestramento preventivo: sono algoritmi che partono dal presupposto che non sia possibile valutare costantemente le azioni dell'algoritmo, per motivi di costo o di difficoltà. Si ricorre quindi ad un addestramento preventivo che deve essere completo. Una volta che il sistema è reso affidabile allora lo si cristallizza, ovvero lo si rende non modificabile. Un esempio sono molti componenti elettronici che utilizzano reti neurali al loro interno. I pesi *sinaptici* di questa rete (ne parleremo ampiamente) non si possono cambiare poiché sono fissati durante la realizzazione del circuito.

3.3 Reti Neurali

3.3.1 Fondamenti biologici e matematici

Fra gli algoritmi di apprendimento supervisionato, quelli derivati dalle reti neurali hanno avuto un ruolo fondamentale nello sviluppo del campo. L'idea che ha permesso di concepire tali reti si trova nel sistema di comunicazioni tra i neuroni.

Come già detto nell'introduzione, gli esseri pluricellulari elaborano le informazioni provenienti dal mondo esterno tramite complesse organizzazioni, la cui unità base è il neurone. Un neurone è una cellula composta da una parte centrale, detta *soma*, da cui si dipartono una fibra nervosa principale, l'*assone*, e una serie di ramificazioni, i *dendriti*. I neuroni sono collegati tra loro attraverso le sinapsi, le quali connettono la parte terminale dell'assone con i dendriti o il soma degli altri neuroni.

3.3.2 Analisi del funzionamento

Una rete neurale artificiale – chiamata normalmente solo rete neurale (o *Neural Network*) – è un modello di calcolo adattivo, capace di cambiare la sua struttura in base alle informazioni esterne (i dati in ingresso) e interne (tramite le connessioni) che scorrono all'interno di essa. Sono strutture non lineari e possono essere usate per simulare relazioni complesse tra ingressi e uscite che altre funzioni analitiche non sarebbero in grado di fare.

L'unità base di questa rete è il *neurone artificiale* introdotto per la prima volta da McCulloch e Pitts nel 1943 (Figura 3.2).

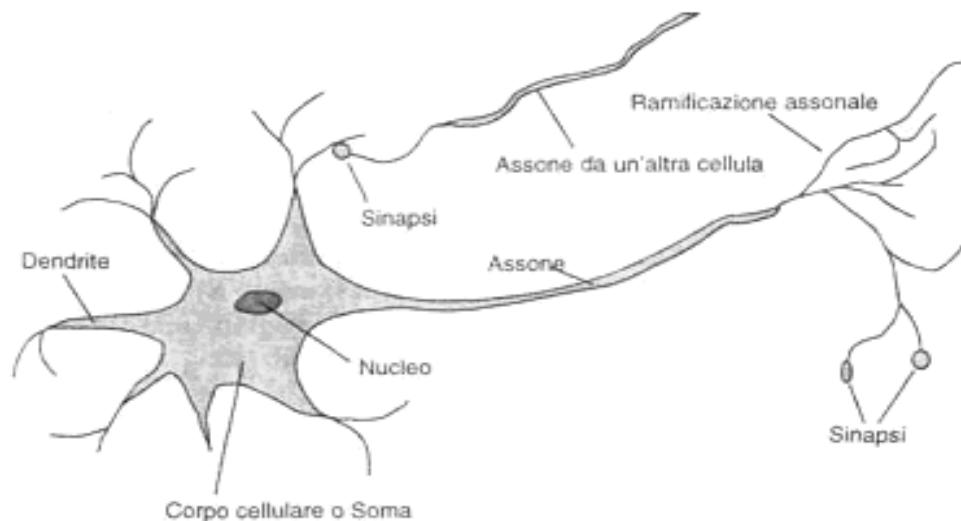


Figura 3.2: schema di un neurone

La comunicazione tra neuroni avviene nel modo seguente. Quando un neurone è attivo invia un impulso elettrico attraverso il suo assone; questo impulso è trasmesso, attraverso le sinapsi, ai neuroni collegati. La trasmissione avviene attraverso lo scambio di sostanze chimiche, chiamate neurotrasmettitori, dall'assone ai recettori presenti sul soma e sulle dendriti. In base al tipo di stimolo e alla configurazione dei recettori, un impulso modifica il potenziale dei neuroni collegati. Ogni neurone ha una propria soglia e esegue una somma pesata degli stimoli che arrivano ai suoi recettori. Se questa somma supera la soglia, il neurone a sua volta si attiva e trasmette un proprio potenziale di attivazione che è trasportato all'assone.

La capacità di apprendere sta nella possibilità di modificare le caratteristiche dei recettori in modo da modificare le configurazioni che comportano l'attivazione del neurone. Modellando le cosiddette *funzioni di attivazione* dei singoli neuroni si può quindi decidere arbitrariamente quale strategia di apprendimento la rete debba seguire.

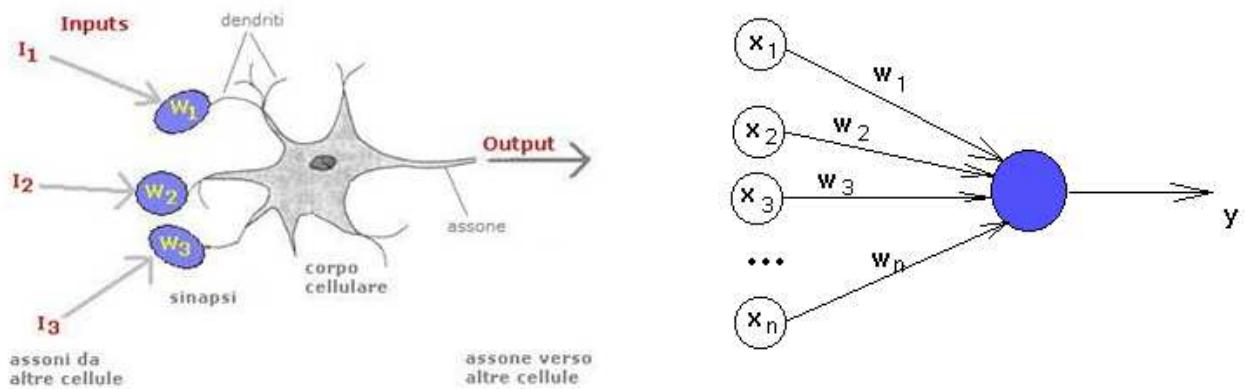


Figura 3.3: Modello di calcolo del neurone (a sinistra) e schema del neurone artificiale (a destra)

Si tratta di un'unità di calcolo a N ingressi e 1 uscita. Come si può vedere dall'immagine a sinistra gli ingressi rappresentano le terminazioni sinaptiche, quindi sono le uscite di altrettanti neuroni artificiali. A ogni ingresso corrisponde un peso sinaptico, che stabilisce quanto quel collegamento sinaptico influisca sull'uscita del neurone.

Si determina quindi il potenziale del neurone facendo una somma degli ingressi, pesata mediante i pesi sinaptici.

A queste è applicata una funzione di trasferimento non lineare:

$$\mathbf{f}(\mathbf{x}) = \mathbf{H}(\sum_i \mathbf{w}_i \mathbf{g}_i(\mathbf{x})) \quad (3.2)$$

Ove $\mathbf{g}_i(x)$ è la funzione in ingresso portata dalle sinapsi e H è una funzione non lineare, solitamente la tangente iperbolica.

Ultimamente però, si discute sul fatto che la *rectified linear unit* o ReLU, rappresenti meglio la modalità di attivazione dei neuroni della corteccia [26]. La ReLU è definita come:

$$\mathbf{f}(\mathbf{x}) = \max(\mathbf{0}, \mathbf{x}) \quad (3.3)$$

È stato verificato da Y. LeCunn et al [7] che scegliere la ReLU è inaspettatamente “*l'elemento singolo più importante di tutta l'architettura per un sistema di riconoscimento*”. Questo può essere dovuto principalmente a 2 motivi:

- la polarità delle caratteristiche è molto spesso irrilevante per riconoscere gli oggetti
- la ReLU evita che quando si fa il pooling (lo vedremo nel capitolo 5) due caratteristiche entrambe importanti ma con polarità opposte si cancellino fra loro

Il modello più elementare di neurone artificiale è il *percettrone* che si comporta in maniera semplificata rispetto a come detto sopra.

La sua uscita è generata in accordo alla seguente:

$$\mathbf{f}(\mathbf{x}) = \mathbf{H}(\langle \mathbf{w}, \mathbf{x} \rangle + \mathbf{b}) \quad (3.4)$$

dove H è una funzione non lineare come detto prima e b è un “bias” ovvero un offset.

La grande peculiarità di queste reti è, come si è visto, la possibilità di combinare più funzioni.

Se queste funzioni fossero sempre lineari non ci sarebbe differenza tra una rete ad uno solo, o a più strati, poiché - come è stato dimostrato in algebra - una composizione di funzioni lineari è a sua volta una funzione lineare. Per questo motivo le non linearità tra un neurone e l'altro sono fondamentali. In tale maniera si riescono a sfruttare reti a più strati creando quello che in letteratura è chiamato *Deep Learning*, di cui si parlerà a breve.

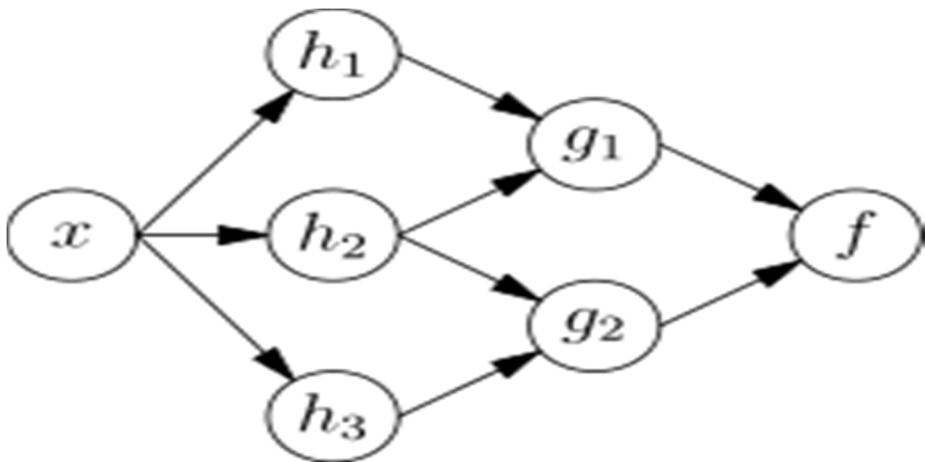


Figura 3.4: rete Feed-Forward. La figura evidenzia la composizione di funzioni

Una rete come quella sopra si chiama rete di tipo *feed-forward*: è cioè un albero aciclico diretto, che si percorre solo nella direzione che va dagli ingressi alle uscite. Come possiamo osservare, a seconda del percorso scelto, si avrà una differente composizione di funzioni.

3.3.3 Tipologia di reti neurali artificiali trattate

Nell'ambito delle reti neurali artificiali sono stati ideati diversi tipi di reti. Saranno fornite l'analisi di due reti in particolare: il *multilayer perceptron* e le *convolutional neural networks*, che sono utilizzate nell'implementazione della rete scelta per classificare le immagini.

Come abbiamo visto nella sezione precedente il percettrone è il tipo più semplice di neurone artificiale. Tuttavia sintetizzare una funzione relativamente semplice, come lo XOR logico, utilizzando un solo neurone non è possibile [8]. Combinando i percetroni in diversi strati completamente collegati fra loro, ognuno dei quali formato da più percetroni in parallelo, si forma una rete detta *multilayer perceptron* (*o MLP*). Questa rete è una rete di tipo *feed-forward* molto utilizzata per l'apprendimento supervisionato

3.3.4 Pregi e difetti

Laddove i normali procedimenti di analisi manuale siano inapplicabili a causa dell'alta complessità dei dati e di elaborazione degli stessi, le reti neurali sono un ottimo strumento poiché possono dedurre funzioni efficaci basandosi solo sull'osservazione di tali dati durante l'apprendimento.

I compiti più tipici per queste reti comprendono: funzioni di regressione e di previsione, di classificazione e riconoscimento delle strutture dei dati, processi decisionali, compressione e filtraggio di dati. Sono quindi ampiamente impiegati nei campi applicativi del machine learning visti nella sezione 3.1, specialmente per quanto riguarda la visione artificiale nella forma delle deep neural network.

Per come sono costruite lavorano in parallelo e sono quindi capaci di trattare molti dati; sono in termini pratici un sofisticato sistema di tipo statistico dotato di una buona immunità al rumore, come vedremo parlando delle convolutional neural networks.

Ad ogni modo i modelli prodotti dalle reti neurali, anche se molto efficienti, non sono sempre spiegabili in maniera comprensibile con il linguaggio simbolico umano: ciò significa che i risultati spesso vanno presi così come sono, da cui anche la definizione inglese delle retineurali come *black box*. A differenza di sistemi algoritmici dove si può esaminare la generazione dell'output passo dopo passo, nelle reti neurali si possono avere anche risultati molto attendibili ma senza capire come e perché siano stati generati.

Non sono presenti inoltre, teoremi per la generazione di reti neurali ottime: la probabilità di ottenere una buona rete sta tutta nelle mani di chi la crea, il quale deve avere dimestichezza con concetti di statistica e particolare attenzione deve dare alla scelta della variabili predittive. Un'ultima cosa da notare, che può a volte essere un problema, è che per essere produttive necessitano di un addestramento che sintonizzi in modo corretto i pesi sinaptici. L'addestramento può richiedere molto tempo se i dati da esaminare e le variabili in gioco sono elevate.

3.4 Multilayer Perceptron

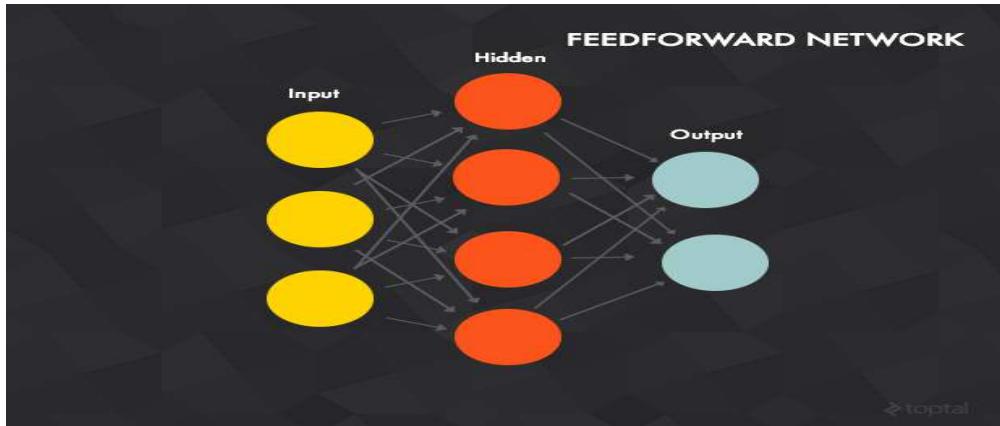


Figura 3.5: Struttura di un classico multiplayer perceptron

3.4.1 Struttura

Il multilayer perceptron (o percepitrone a strati multipli) è un tipo di rete feed-forward che mappa un set di input ad un set di output. È la naturale estensione del percepitrone singolo e può permettere di distinguere dati non linearmente separabili.

Il mlp possiede le seguenti proprietà:

- ogni neurone (o unità, i termini sono intercambiabili) è un percepitrone come quello descritto nella sezione. Possiede quindi una propria funzione d'attivazione non lineare.
- a ogni connessione tra due neuroni corrisponde un peso sinaptico w .
- è formato da 3 o più strati: ciò lo rende una rete neurale profonda (deep neural network). Nella figura 3.4 sono mostrati 1 strato di input, un cosiddetto “*hidden layer*” (uno strato nascosto) e uno strato di output.
- l'uscita di ogni neurone dello strato precedente è l'ingresso per ogni neurone dello strato successivo. È quindi una rete *fully connected*. (Tuttavia si possono disconnettere selettivamente settando il peso sinaptico w a 0).
- la dimensione dell'input e la dimensione dell'output dipendono dal numero di neuroni di questi due strati. Il numero di neuroni dello strato nascosto è indipendente.

Guardando la figura 3.5, l'elaborazione dei dati avviene assegnando il primo strato dei neuroni con i valori dell'input – in questo caso la rete può elaborare input tridimensionali – i quali saranno elaborati e propagati ai successivi strati fino ad avere in uscita, un output di dimensioni pari a quattro.

Il primo e l'ultimo strato devono avere un numero di neuroni pari alla dimensione dello spazio di ingresso e quello di uscita. Queste sono le terminazioni della black box che rappresenta la funzione che vogliamo approssimare.

3.4.1.1 Strati Nascosti

I cosiddetti *hidden layers* sono una parte molto interessante della rete. Per il teorema di approssimazione universale [29], una rete con un singolo strato nascosto e un numero finito di neuroni, può essere addestrata per approssimare una qualsiasi funzione. In altre parole, un singolo strato nascosto è abbastanza potente da imparare qualsiasi funzione.

Detto ciò, nella pratica si può arrivare ad avere migliori risultati su una rete con più strati nascosti, una rete cioè più profonda poiché come vedremo nella sezione del deep learning gli strati nascosti sono gli strati dove la rete memorizza la propria rappresentazione astratta dei dati che riceve in ingresso.

Esempio: le immagini

Supponiamo che nell'immagine in ingresso ci sia un autobus. Il primo strato, nel caso fosse un edge-detector, potrebbe trovare i bordi nell'immagine direttamente dai pixel *raw*. Questo però non basta nella stragrande maggioranza dei casi per sapere se quel bordo appartiene a una faccia, ad un autobus o ad un elefante. Diciamo ora che abbiamo a disposizione un secondo strato – uno strato nascosto appunto – in cui ogni neurone è capace di riconoscere un oggetto diverso dall'altro neurone. Mettendo insieme i bordi identificati dal primo strato, si ottiene l'attivazione del neurone sensibile alle caratteristiche dell'autobus e quindi in uscita si ha l'informazione che nella foto in ingresso è stato identificato un autobus.

3.4.2 Apprendimento mediante Back-Propagation

Il mlp utilizza una tecnica di apprendimento supervisionato nota come algoritmo di *back-propagation of errors*. Inizialmente un esempio del training set è presentato alla rete e propagato fino alla fine. Dopodiché l'algoritmo di backpropagation permette di calibrare i pesi sinaptici in base alla discrepanza tra la risposta desiderata e la risposta effettivamente data dalla rete, mediante la tecnica della “discesa del gradiente” applicata alla funzione di costo, la quale molto spesso è lo scarto quadratico. (Vedere sezione 3.2).

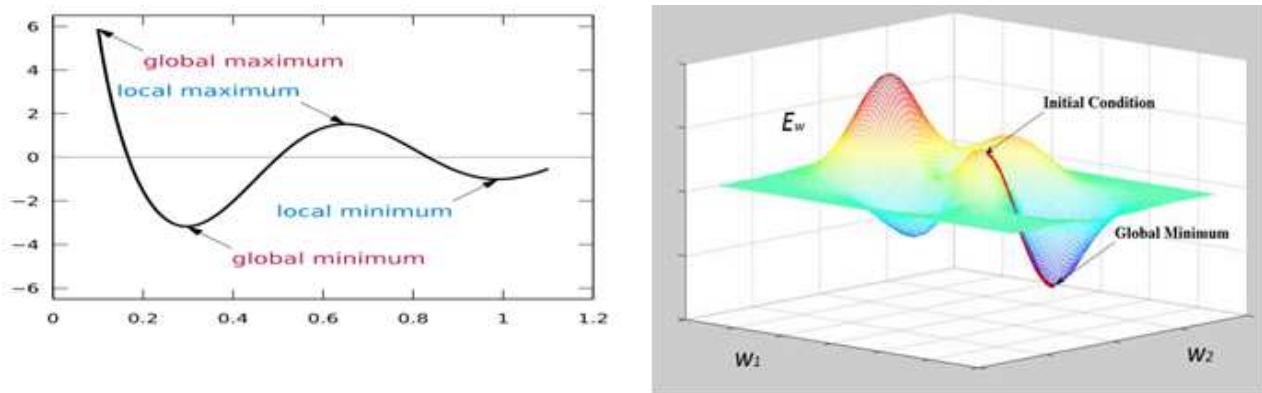


Figura 3.6: Discesa del gradiente in 2 e in 3 dimensioni [33]

Il valore migliore di ogni peso sinaptico della rete è quello per cui il gradiente della funzione di costo raggiunge il minimo globale. Durante la fase di apprendimento i pesi vengono cambiati volta per volta, cercando sempre di aggiornarli verso la direzione in cui il gradiente diminuisce. Questo non è però una cosa facile perché nelle reti questo minimo dipende da *tutti* i pesi della rete quindi ci sono molte variabili in gioco.

La discesa del gradiente applicata ai problemi di machine learning rimase sconosciuta fino alla scoperta dell'algoritmo di back propagation da parte di Hinton, Rummelhart e Williams [8], che permette di calcolare il gradiente di sistemi non lineari a livelli multipli. L'idea è quella di poter calcolare in modo efficiente il gradiente dell'errore gradualmente dall'output all'input.

3.5 Deep Learning

Il Deep Learning è un ramo specifico del machine learning che, partendo da dati grezzi, cerca di ottenere una maggiore astrazione di questi mediante un modello strutturato a più livelli (o layers), nei quali avanzando, si va appunto più nel “profondo” e si astrae sempre di più; i concetti ai livelli più profondi sono definiti a partire da quelli ai livelli precedenti tramite una serie di trasformazioni non lineari.

Questa architettura è biologicamente ispirata al cervello dei mammiferi, il quale è organizzato in modo *profondo e gerarchico*, dove un certo dato in ingresso viene rappresentato da più livelli di astrazione, ognuno dei quali corrisponde ad una diversa area della corteccia. Il cervello sembra anche elaborare le informazioni attraverso più fasi di trasformazione e rappresentazione. Ne è un esempio il sistema visivo primate, il quale le elabora tramite diverse fasi come il rilevamento dei bordi, la percezione delle forme, da quelle primitive a quelle gradualmente sempre più complesse. Un altro esempio possono essere i bambini che, prima di capire il significato complesso di una parola, devono costruirlo in base ad un apprendimento gerarchico, composto ad esempio dall'insieme dei suoni, dei gesti e del contesto in cui viene emesso quel determinato suono.

Il deep learning ha origini dagli studi sul “Neocognitron” di K. Fukushima nel 1980. Nove anni dopo Yann LeCunn riesce ad applicare l'algoritmo di back propagation alle deep neural network con lo scopo di riconoscere gli zip code scritti a mano sulla posta. Nonostante il successo del suo lavoro la rete richiedeva all'incirca 3 giorni di allenamento allora, rendendola una tecnologia poco pratica. Con l'avvento delle support vector machines, che erano modelli di rete più semplici e veloci, le deep neural network vengono momentaneamente accantonate. Nella metà degli anni 2000 è stato scoperto un metodo per allenare queste reti profonde strato per strato, rendendole molto più veloci e accurate nel calibrare i pesi sinaptici. Da allora hanno sempre dato risultati sorprendenti, soprattutto nella forma delle Convolutional Neural Networks e soprattutto nei compiti di visione artificiale dove surclassano ogni altro tipo di approccio con margini spesso anche molto elevati.

Un'altra caratteristica che ne ha favorito il ritorno è quella del notevole miglioramento hardware degli ultimi anni. In particolare, le GPU sono molto indicate per i calcoli matriciali in più dimensioni in parallelo tipici dei problemi di machine learning.

Questo approccio “*Deep*” è stato proposto in vari problemi riguardanti la visione artificiale, elaborazione del linguaggio naturale e riconoscimento vocale ottenendo risultati che lo classificano di dovere come tecnologia allo stato dell'arte.

Riconducendoci alla visione artificiale e al problema della classificazione, posto tra gli obiettivi primari di questa tesi, possiamo capire come un'elaborazione a livelli possa essere efficace su input di tipo visivo. I vari strati della rete infatti funzionano da estrattore di caratteristiche dell'immagine in ingresso.

Il primo strato della rete potrà, analizzando l'immagine, cogliere delle caratteristiche particolari ma non sarà ancora in grado di attribuirle un significato semantico per poter riconoscere in essa un oggetto specifico o una faccia, per esempio. Nell'attraversare la rete queste caratteristiche “elementari” sono combinate insieme, donando ad ogni strato una capacità di astrazione maggiore di quello precedente, fino ad arrivare – nel migliore dei casi – ad attribuire il corretto significato a queste combinazioni di caratteristiche (es. bordi → parti di un oggetto → oggetto). Le prestazioni e la precisione del riconoscimento possono variare molto secondo il numero e delle connessioni presenti tra i vari strati di questa architettura.

Un esempio per spiegare l'efficacia del modello di rete profonda a più livelli è spiegato nella sottosezione 3.4.1.1, riguardante gli hidden layers.

4. CONVOLUTIONAL NEURAL NETWORKS

4.1 Storia

Le reti neurali tradizionali completamente connesse, di dimensioni sufficientemente grandi, potrebbero imparare a riconoscere le immagini senza bisogno di estrarre le features. Tuttavia un approccio del genere potrebbe ad un numero di variabili elevatissimo e quindi ad un addestramento praticamente ingestibile.

Le Convolutional Neural Networks, d'ora in poi chiamate CNNs, sono delle deep neural networks con una particolare architettura estremamente vantaggiosa per compiti di tipo visivo. Sono state ispirate dalle ricerche biologiche di Hubel e Wiesel i quali, come accennato nel capitolo 2, studiando il cervello dei gatti, avevano scoperto che la loro corteccia visiva conteneva una complessa struttura di cellule. Queste cellule erano sensibili a piccole parti locali del campo visivo, detti campi recettivi (receptive fields). Questi filtri locali sono perfetti per comprendere la correlazione locale degli oggetti in un'immagine.

Vi sono cellule semplici (S), sensibili ai bordi nel loro campo recettivo, e cellule complesse (C) che sono *localmente invarianti* rispetto allo stimolo. Essendo questi sistemi i più efficienti in natura per la comprensione e l'elaborazione delle immagini, è naturale che i ricercatori abbiano tentato di simularli. Come vedremo, le cellule S vengono simulate dallo strato di convoluzione, le cellule C da quello di pooling.

Durante un periodo di ricerca proprio agli AT&T Bell Laboratories (gli stessi dove è stato ideato il metodo a SVM, capitolo 2), Yann LeCun migliora il modello[1] proposto da Fukushima e sviluppa le CNN.

4.2 Architettura

4.2.1 Panoramica Generale

Le CNNs sono deep neural networks costituite da diversi strati che fungono da image processing ed un classificatore finale, solitamente un normale MLP oppure una SVM.

Questi strati in cui si estraggono le caratteristiche delle immagini sono detti *convolutional layers*, e sono generalmente seguiti da una funzione non lineare e un passo di subsampling, per cui i passi per ogni strato sono: convoluzione, funzione non lineare, subsampling.

Convoluzione e subsampling hanno come scopo quello di estrarre le caratteristiche, mentre l'unità non lineare serve a rafforzare le caratteristiche più forti e indebolire quelle meno importanti, quello che cioè hanno stimolato meno i neuroni (si dice che fa da "squashing").

Abbiamo visto nel capitolo 3 le funzioni più utilizzate e più efficaci, come la tangente iperbolica e la ReLU.

Possiamo inoltre notare che, per ogni immagine in input, corrispondono nei vari strati, diversi gruppi di immagini, chiamati *feature maps*. Le feature maps sono il risultato dell'operazione di convoluzione svolta tramite dei filtri, chiamati anche *kernel*, che in questo caso non sono altro che delle matrici con dei valori utili a ricercare determinate feature nelle immagini.

Infine, terminati i convolutional layers, le feature maps vengono "srotolate" in vettori e affidate ad una rete neurale classica che esegue la classificazione finale.

Vi possono essere un numero arbitrario di convolutional layer, solitamente 2-3 per i problemi di classificazione standard. Poco tempo fa, Alex Krizhevsky et al [13] dell'Università di Toronto hanno allenato – tramite ausilio di GPU Tesla K40 – una rete costituita da 5 convolutional layers, 60 milioni di parametri e 650 mila neuroni. Hanno ottenuto la migliore percentuale d'errore al mondo sul dataset ImageNet ILSVRC-2010, contenente 1,2 milioni di immagini divise in 1000 categorie. Vedremo che, per il problema affrontato nel corso della tesi, si è scelto di utilizzare una CNN con 2 convolutional layers.

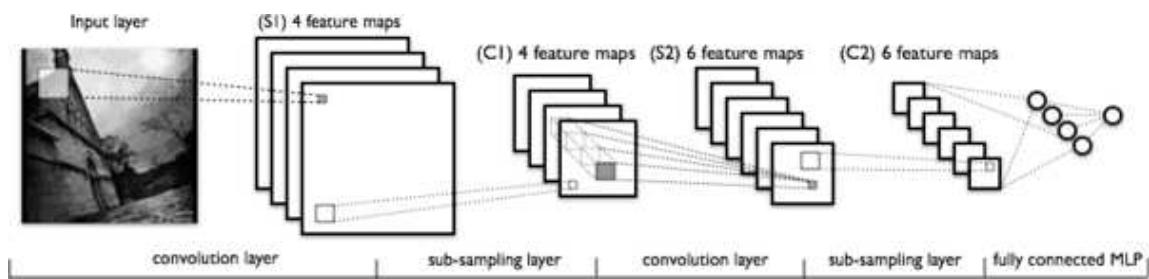


Figura 4.1 : Architettura di una CNN tradizionale [28]

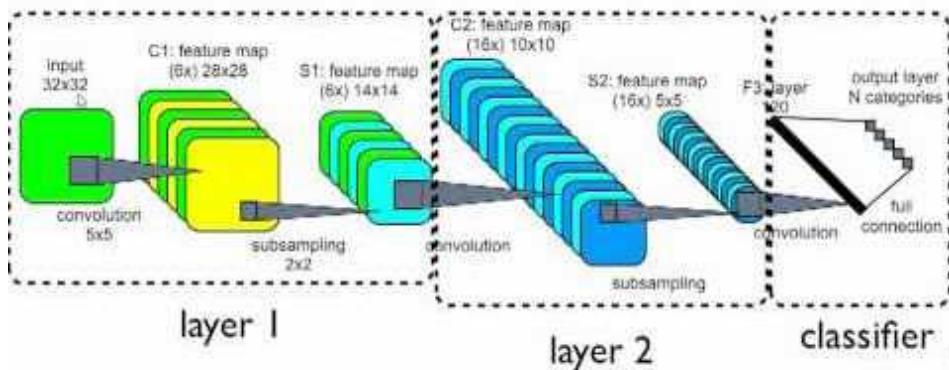


Figura 4.2: Architettura CNN : divisione in layer [35]

4.3.2 Convoluzione

Per prima cosa definiamo cos'è, e come si applica un filtro ad un'immagine digitale. Un'immagine digitale può essere considerata come una matrice A di dimensione $M \times N$ valori reali o discreti. Ogni valore della matrice prende il nome di pixel e i suoi indici sono anche chiamati coordinate: ogni pixel $A(m,n)$ rappresenta l'intensità nella posizione indicata dagli indici.

Si definisce “filtro” o “kernel” una trasformazione applicata ad un’immagine. Possiamo definire vari tipi di filtri in base a quale regione dell’immagine di partenza sia necessaria per ottenere il valore di un pixel:

- Globali: per determinare il valore dell’immagine di output è necessario conoscere ogni pixel dell’immagine di ingresso.
- Puntuali: il valore di un pixel dipende solo da un altro pixel, in genere quello che ha le stesse coordinate del pixel da calcolare.
- *Locali*: il valore di un pixel dell’immagine di uscita dipende dal valore dei pixel di una sottoregione dell’immagine di ingresso. Se tale sottoregione ha lato pari a L, il filtro sarà una matrice quadrata di dimensione LxL e il valore di ogni pixel dell’immagine di uscita sarà quindi ottenuto da una funzione.

$$\mathbb{R}^{L \times L} \rightarrow \mathbb{R}$$

Riguardo ai filtri globali e locali, un particolare tipo di operazione che si può eseguire per è rappresentata dalla convoluzione. La convoluzione, discreta nel caso di immagini digitali, si può definire come:

$$y[m, n] = x[m, n] * h[m, n] = \sum_{j=-\infty}^{\infty} \sum_{i=-\infty}^{\infty} x[i, j] \times h[m - i, n - j] \quad (4.1)$$

dove $y[m, n]$ è l’uscita, $x[m, n]$ è l’immagine in ingresso e $h[m, n]$ la matrice di convoluzione.

Ogni pixel di $y[m, n]$ è così il risultato di una somma pesata tramite $h[m, n]$ della sottoregione che ha centro nel pixel indicato dalle coordinate m, n .

È importante considerare che per gli estremi di $x[m, n]$ la matrice di convoluzione tocca zone non definite. Nelle CNN si sceglie quindi di ignorarle e ottenere in output un’immagine che ha dimensioni inferiori. Precisamente, se l’immagine di ingresso ha dimensioni $M \times N$ e il kernel $m \times n$, in uscita si avrà un’immagine di dimensioni: $(M - m + 1) \times (N - n + 1)$.

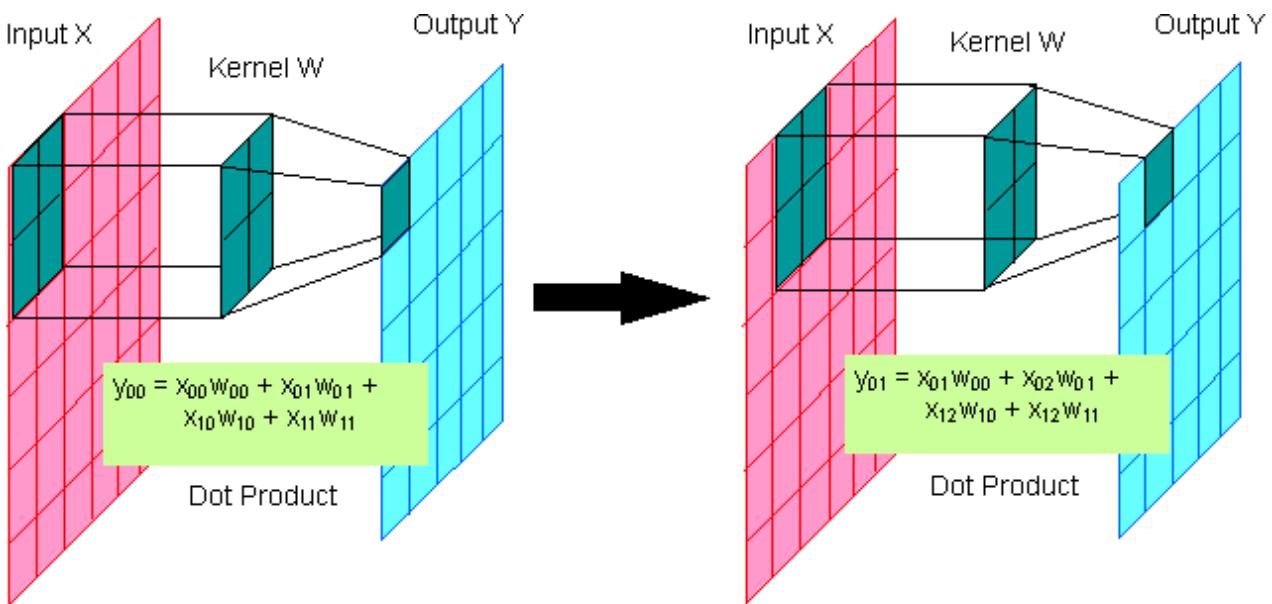


Figura 4.3: Convoluzione con un kernel: primi due step

Nei convolutional layers viene fatta un'operazione di convoluzione tra l'immagine/i in ingresso e un numero arbitrario di filtri. L'insieme di questi filtri viene chiamato filter bank. Questi filtri hanno valori tali da ottenere in uscita – tramite convoluzione – un riconoscimento di determinate caratteristiche. Ci possono essere filtri che evidenziano i bordi verticali, orizzontali, a 45°, e così via (a titolo di esempio si citano i filtri di Gabor, sezione 2.2.2.4).

I valori dei kernel sono all'inizio scelti casualmente, e vengono poi migliorati ad ogni iterazione mediante l'algoritmo di back propagation. Così facendo la rete addestra i suoi filtri ad estrarre le features più importanti degli esempi del training set; cambiando training set i valori dei kernel saranno diversi.

Ad esempio, i valori dei kernel di una rete allenata con immagini di pali verticali saranno diversi da quella allenata con immagini di palloni da calcio, dovuto al fatto che nel primo caso i valori saranno opportunamente calibrati per riconoscere lunghi orientamenti verticali, mentre nel secondo per riconoscere i confini sferici. Il suddetto addestramento accade per i filtri di ogni strato della rete.

Nei convolution layers ci sono N immagini in ingresso che vengono convoluti con K filtri. Ciascun filtro produce quindi N immagini, una per ogni immagine proveniente dallo strato precedente *. Queste immagini vengono sommate e il risultato è una delle feature maps prodotte dai diversi filtri. Possiamo vederne un esempio nella figura 4.1, nel secondo strato di convoluzione: il pixel della feature map di S2 è composto dalla convoluzione del filtro con le 4 feature maps di C1. Le feature maps vengono poi fatte passare attraverso un livello di subsampling ed una funzione non lineare e saranno poi l'input per un successivo convolutional layer.

Il primo strato di convoluzione si occupa di estrarre features direttamente dai pixel dell'immagine e li memorizza nelle feature maps.

Questo output diverrà poi l'input di un successivo livello di convoluzione (dopo essere passato dal subsampling) il quale andrà a fare una seconda estrazione delle caratteristiche, questa volta dall'output del primo livello e non direttamente dai pixel, ottenendo una maggior astrazione e comprensione delle features. I vari strati della rete sono degli hidden layers, come abbiamo visto nel capitolo 3, mentre le feature maps possiamo vederle come dei buffer dove memorizzare le features estratte prima di farle passare attraverso un'altra elaborazione.

Un'interessante proprietà degli strati di convoluzione sta nel fatto che se l'immagine in input viene traslata, l'output della feature map sarà traslato della stessa quantità ma rimarrà invariato altrove. Questa proprietà è alla base della robustezza rispetto alle traslazioni e alle distorsioni dell'immagine in ingresso. La figura 4.4 evidenzia gli effetti sull'estrazione delle features nei vari passaggi della rete.

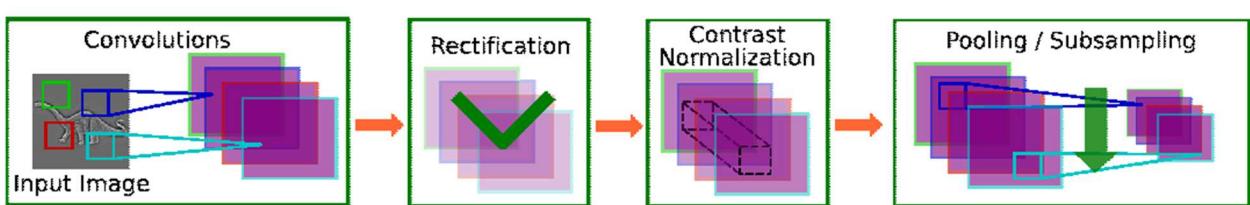


Figura: 4.4 Architettura CNN: Features extraction[9]

*Vedremo che in realtà i collegamenti tra gli input e i filtri si impostano in maniera diversa e più efficace

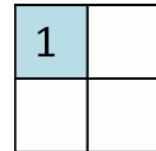
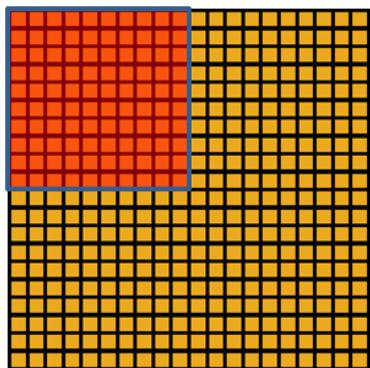
4.2.3 Subsampling

Un'altra proprietà che si vuole ottenere per migliorare i risultati sulla visione artificiale è il riconoscimento delle features indipendentemente dalla posizione nell'immagine, perché l'obiettivo è quello di riconoscere le feature in ogni punto e rafforzare l'efficacia contro le traslazioni e le distorsioni. Questo si può ottenere diminuendo la risoluzione spaziale dell'immagine, il che favorisce anche una maggiore velocità di computazione.

Lo strato di subsampling ottiene in ingresso N immagini di una risoluzione e restituisce in uscita sempre N immagini, ma con una risoluzione diminuita in rapporto alla grandezza dei pool del subsampling, come abbiamo visto nel capitolo 3. Spesso viene utilizzato un pool di 2x2, il che significa che si dividerà l'immagine in zone di 2x2 non sovrapposte e per ogni zona viene scelto un solo pixel; quindi in uscita avremo un'immagine di $\frac{1}{4}$ della risoluzione precedente.

I criteri con cui scegliere il pixel sono diversi:

- average pooling: si calcola il valore medio sui pixel del pool
- median pooling: si calcola la mediana dei valori dei pixel del pool
- LP-pooling: si calcola la p-norma della matrice dei pixel a seconda del valore di p.
Yann LeCun et al[11] hanno verificato che sul dataset SVHN si hanno prestazioni migliori per valori di $p=2, 4, 12$.
- max pooling: dei pixel passa solo quello col valore più alto. È un'operazione di lp-pooling con $p = \infty$



Convolved feature Pooled feature

Figura 4.4: pooling del primo quarto di immagine

Attraversando la rete si avrà gradualmente:

- > Un aumento del numero di feature maps e quindi della ricchezza della rappresentazione delle features.
- > Una diminuzione della risoluzione dell'input.

Questi fattori combinati insieme donano un forte grado di invarianza alle trasformazioni geometriche dell'input.

4.3 Pregi e peculiarità

L'estrazione delle features avviene, come abbiamo visto, tramite la convoluzione con dei kernel. I kernel possono essere visti come dei campi recettivi locali, che cercano una feature in un “intorno” dato dalla loro dimensione. Per come è definita la convoluzione tra matrici, il kernel effettua una ricerca della stessa feature su tutta l'immagine forzando le unità di una feature map ad avere gli stessi pesi; si dice che le unità hanno pesi condivisi (*shared weights*). Questo si traduce in molti meno parametri da allenare per la rete, giovando al processo di addestramento.

Ricordiamo che tra due diverse feature maps invece, i pesi sono diversi, in quanto ognuna è dedita all'estrazione di una specifica caratteristica. Queste idee architettonali rendono le CNN degli strumenti che richiedono un basso contributo di pre-processing dell'immagine in ingresso.

I receptive fields sono le connessioni tra i vari strati della CNN e la maniera di collegarli è un parametro molto importante ai fini delle performance.

È stato dimostrato più volte che filtri casuali hanno performance solo leggermente più basse rispetto a filtri finemente addestrati [13] suggerendo che i filtri, usati in ogni strato di convoluzione, *non* giocano un ruolo principale riguardo alle performance.

I veri responsabili del grande successo delle CNN sono le connessioni tra gli strati e le combinazioni delle features [14]. È stato dimostrato in diversi articoli (come quelli di Sermanet et al, e Hadsell et al [28]) che è preferibile avere una ricchezza di feature rispetto alla densità delle connessioni.

Sebbene sia una ricerca sulle modalità di connessioni per migliorare l'efficienza di un apprendimento *non* supervisionato, E. Culurciello et Al [15] hanno verificato che, in generale, avere i filtri dei vari strati completamente connessi peggiora la specificità degli stessi, rendendoli più uniformi e meno specifici ed efficaci nel trovare una determinata caratteristica rispetto ad un'altra. La densità delle connessioni è inoltre motivo di rallentamento delle

prestazioni, come ho potuto verificare io stesso nell'addestramento delle varie CNN con Torch.

Possiamo notare nella fig. 5.4 tre esempi di tecniche di connessione. A sinistra una connessione con un fan-In di ogni neurone pari a 1. Nel mezzo abbiamo $\text{fan-In} = 2^*$ e nell'ultimo esempio una rete completamente connessa. In basso nella stessa abbiamo due tavole di filtri. Quelli a sinistra sono stati ottenuti con un collegamento completo mentre quelli a destra con un $\text{fan-In}=2$. Si nota chiaramente che quelli a sinistra siano più uniformi e quindi meno efficaci nel trovare specifiche features. [15]

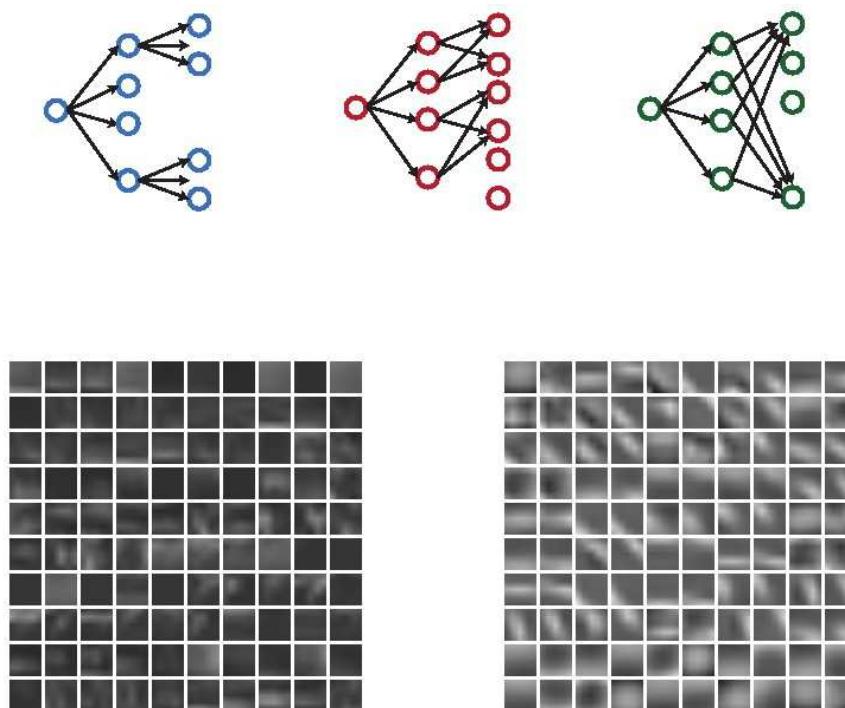


Figura 4.5: Collegamenti ed effetti sui filtri [15]

Le connessioni solitamente vengono scelte in modo random rispettando il fan-In, e cercando di avere un numero uniforme di connessioni da parte di ogni neurone dello strato precedente.

4.4 Utilizzi e prestazioni

L'alta efficacia, il ridotto numero di variabili in gioco, e tutte le altre peculiarità insieme con l'enorme progresso tecnologico dell'hardware, hanno reso le CNN il sistema più promettente per compiti di riconoscimento dell'immagine, con moltissimi ambiti applicativi come: robot, automobili autonome, smartphone, sistemi di sicurezza e altri.

Recentemente le CNN sono state implementate su FPGA con risultati sorprendenti e si mira a costruire coprocessori con interfacce ARM da poter inserire sullo stesso package per portare questa tecnologia ad essere nativamente supportata su smartphone e sistemi embedded [16].

Parlando di numeri, le CNN hanno raggiunto un errore di 0.23% sul popolare database di cifre scritte a mano MNIST [17] nel Febbraio 2012, record tutt'ora imbattuto. In un altro articolo hanno raggiunto il 97.6% di risposte esatte sul riconoscimento di oggetti fermi in più di 5600 immagini in 10 classi [33].

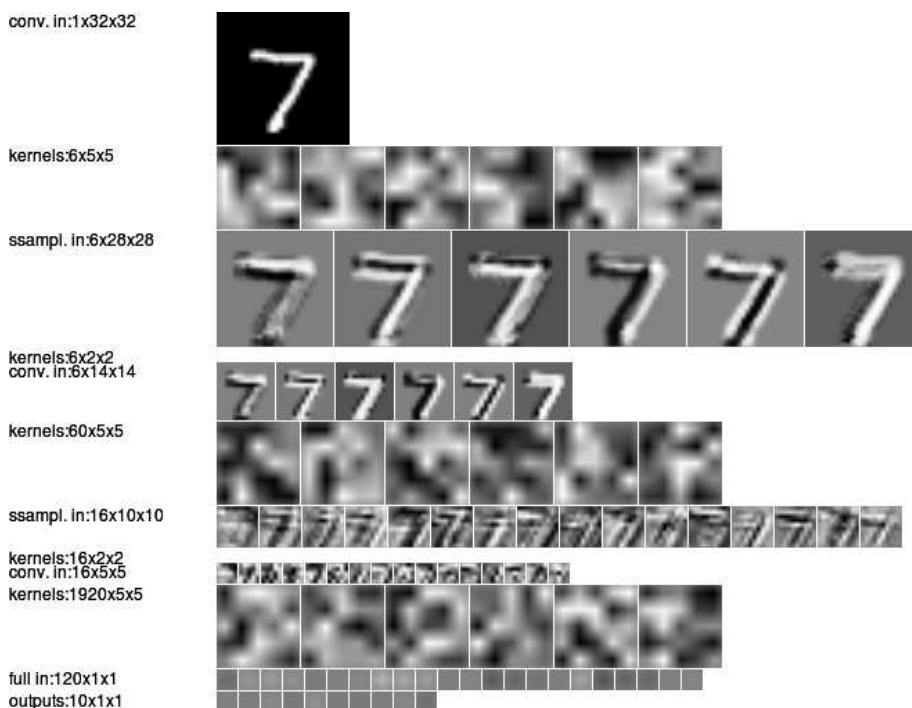


Figura 4.6: Riconoscimento di cifre del dataset MNIST : passi della CNN [yann lecun]

In una recente ricerca [18], il dipartimento di visione artificiale del Royal Institute of Technology di Stoccolma ha utilizzato "*OverFeat*", una CNN pubblica allenata per ILSVRC13, una competizione annuale di riconoscimento degli oggetti. Il dataset di ImageNet ILSVRC13 include 1000 classi e 1.2 milioni di immagini per il training, validation e testing.

OverFeat è una CNN con un classificatore di tipo SVM alla fine. L'articolo sottolinea come loro abbiano utilizzato questa CNN "off-the-shelf" ovvero già pronta e, senza allenarla ulteriormente, l'hanno testata contro altri metodi allo stato dell'arte finemente perfezionati finora sviluppati. Come test hanno scelto attività gradualmente sempre più lontane dall'originario compito per cui *OverFeat* è stata allenata e, con enorme stupore, hanno verificato che *OverFeat* surclassa i suddetti metodi su qualsiasi dataset (si rimanda all'articolo per i dettagli) nonostante sia stata allenata solo mediante l'ImageNet.

OverFeat è utilizzato non solo per la classificazione. Per quanto riguarda l'object-detection [19] risulta avere senza problemi prestazioni, in termini di accuratezza, migliori del 10% rispetto al metodo migliore prima esistente. Anche nel Visual Instance Retrieval viene confrontata con i migliori metodi in circolazione [20], i quali hanno anche il vantaggio di essere stati accuratamente calibrati per i relativi dataset di test (al contrario di *OverFeat*), ottenendo l'ennesima conferma.

L'articolo si chiude con una frase che qui cito:

"Thus, it can be concluded that from now on, deep learning with CNN has to be considered as the primary candidate in essentially any visual recognition task."

Method	mean Accuracy
HSV [27]	43.0
SIFT internal [27]	55.1
SIFT boundary [27]	32.0
HOG [27]	49.6
HSV+SIFTi+SIFTb+HOG(MKL) [27]	72.8
BOW(4000) [14]	65.5
SPM(4000) [14]	67.4
FLH(100) [14]	72.7
BiCos seg [7]	79.4
Dense HOG+Coding+Pooling[2] w/o seg	76.7
Seg+Dense HOG+Coding+Pooling[2]	80.7
CNN-SVM w/o seg	74.7
CNNaug-SVM w/o seg	86.8

	Dim	Oxford5k	Paris6k	Sculp6k	Holidays	UKBench
BoB [3]	N/A	N/A	N/A	45.4 [1]	N/A	N/A
BoW	200k	36.4 [20]	46.0 [35]	8.1 [3]	54.0 [4]	70.3 [20]
IFV [3]	2k	41.8 [20]	-	-	62.6 [20]	83.8 [20]
VLAD [4]	32k	55.5 [4]	-	-	64.6 [4]	-
CVLAD [3]	64k	47.8 [2]	-	-	81.9 [2]	89.3 [2]
HE+burst [17]	64k	64.5 [42]	-	-	78.0 [42]	-
AHE+burst [7]	64k	66.6 [42]	-	-	79.4 [42]	-
Fine vocab [26]	64k	74.2 [26]	74.9 [26]	-	74.9 [26]	-
ASMK*+MA [12]	64k	80.4 [42]	77.0 [42]	-	81.0 [42]	-
ASMK+MA [42]	64k	81.7 [42]	78.2 [42]	-	82.2 [42]	-
CNN	4k	32.2	49.5	24.1	64.2	76.0
CNN-ss	32-120k	55.6	69.7	31.1	76.9	86.9
CNNaug-ss	4-15k	68.0	79.5	42.3	84.3	91.1
CNN+BOW [6]	2k	-	-	-	80.2	-

Figura 4.7: Alcuni risultati riportati in [18]. A sinistra riconoscimento d'immagini, a destra Vis. Instance Retrieval

4.5 Confronto con l'uomo

Nel 2011, le CNN hanno la prima volta battuto l'uomo raggiungendo un errore di 0.56% contro l'1.16% degli umani sul riconoscimento dei segnali stradali nella competizione “German Traffic Sign competition run by IJCNN 2011” [37].

Nel 2013 è stata introdotta una CNN a N dimensioni che ha una rappresentazione interna molto potente: consente di elaborare dati a N dimensioni e al contempo avere una adeguata interpretazione delle caratteristiche peculiari dell'oggetto. I risultati sperimentali hanno dimostrato che nella classificazione facciale 3D per genere ha battuto l'uomo con una percentuale di addirittura il 33%.

Questi risultati bastano per comprendere la potenzialità delle convolutional neural networks e per dare un'idea di numerosi possibili campi applicativi.

5. IL FRAMEWORK TORCH

5.1 Introduzione

Torch [2] è un framework per il calcolo numerico versatile ed è una libreria che estende il linguaggio Lua. L'obiettivo è quello di fornire un ambiente flessibile per il progetto e addestramento di sistemi di machine learning anche su larga scala.

La flessibilità è ottenuta mediante Lua, un linguaggio di scripting estremamente leggero e efficiente. Le prestazioni sono garantite da backend compilati ed ottimizzati (C,C++,CUDA,OpenMP/SSE) per le routine di calcolo numerico di basso livello.

Gli obiettivi degli autori erano: (1) facilità di sviluppo di algoritmi numerici; (2) facilità di estensione, incluso il supporto ad altre librerie; (3) la velocità.

Gli obiettivi (2) e (3) sono stati soddisfatti tramite l'utilizzo di Lua poiché un linguaggio interpretato risulta conveniente per il testing rapido in modo interattivo, garantisce facilità di sviluppo e, grazie alle ottime C-API, unisce in modo eterogeneo le varie librerie, nascondendole sotto un'unica struttura di linguaggio di scripting e mantenendo tutte le funzionalità. Infine, essendo ossessionati dalla velocità, è stato scelto Lua poiché è un veloce linguaggio di scripting e può inoltre contare su di un efficiente compilatore JIT.

Inoltre, Lua ha il grosso vantaggio di essere stato progettato per essere facilmente inserito nelle applicazioni scritte in C e consente quindi di "wrappare" le sottostanti implementazioni in C/C++ in maniera banale. L'interfaccia per il Lua è tra le più semplici e dona quindi grande estensibilità al progetto Torch.

Torch è un framework auto-contenuto e estremamente portabile su ogni piattaforma: iOS, Android, FPGA, processori DSP ecc. Gli script che vengono scritti per Torch riescono ad essere eseguiti su queste piattaforme senza nessuna modifica.

Per soddisfare il requisito (1) hanno invece ideato un oggetto chiamato *Tensor*, il quale altro non è che una "vista" di una particolare area di memoria, e permette una efficiente e semplice

gestione di vettori a N dimensioni, tensori appunto. Diversi Tensor possono puntare alla stessa area di memoria ma avere una vista geometrica diversa delle stessa. Vi sono tensori di diverso tipo ma le principali operazioni matematiche sono definite per il `doubleTensor` e il `floatTensor`.

```
1 floatT = torch.FloatTensor(100,100) [[crea un tensore in virgola
2 mobile a precisione singola]]
3 randomT = torch.rand(100,100) -- immette valori random
4 r = floatT + 1/2 -- operatori di base
5 r:add(0.5, floatT) -- operatori in-place
6 floatT:fill(1) --riempie il tensore di 1
7 [[ In Lua i ':' aggiunge semplicemente un primo
8 argomento alla funzione chiamato self.
9 Questo viene utilizzato per usare funzioni sull'oggetto
10 stesso e permette quindi di usare tali oggetti
11 come classi.
12 in sostanza è come chiamare un metodo ]]
13
14 doubleT = floatT:double() [[casting a tensore a
15 dobbia precisione]]
16 r = torch.FloatTensor(doubleT:size()) --stesso size di doubleT
17 r:copy(doubleT) -- cast automatico double -> float
18
```

Figura 5.1: esempi di Utilizzo di Tensor

L'oggetto Tensor fornisce anche un'efficiente gestione della memoria: ogni operazione fatta su un oggetto Tensor non alloca nuova memoria, ma trasforma il tensore esistente o ritorna un nuovo tensore che referenzia la stessa area di memoria. Se si vuole davvero avere una copia dell'area di memoria esistono i metodi `Tensor:copy()` e `Tensor:clone()`.

Torch fornisce un ricco set di routine di calcolo numerico: le comuni routine di Matlab, algebra lineare, convoluzioni, trasformata di Fourier ecc.

Ci sono molti package utili a diverse categorie: Machine Learning, Visione Artificiale, Image processing, Video, Speech Recognition, ecc.

I package più importanti per il machine learning sono:

- **nn**: Neural Network, fornisce ogni sorta di modulo per la costruzione di reti neurali, reti neurali profonde (deep), regressione lineare, MLP, autoencoders ecc. È il package che ho utilizzato per lo sviluppo della CNN.
- **optim**: package per l'ottimizzazione di algoritmi di back propagation. Fondamentale per avere buone performance nel training della rete.
- **unsup**: è un toolbox per l'apprendimento *non* supervisionato. Non è stato utilizzato, in quanto per il nostro problema abbiamo scelto un apprendimento supervisionato.
- **Image**: package per l'image processing.
- **cunn**: package per utilizzare le reti neurali sfruttando la potenza di calcolo parallelo delle GPU, mediante l'architettura CUDA.

Sono disponibili molti altri package, per installarli si ricorre al package manager di Lua: `luarocks`.

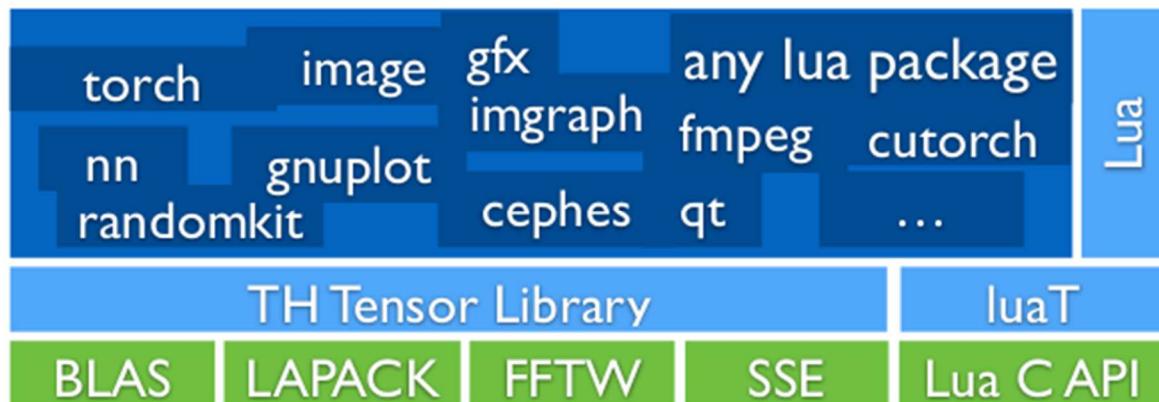


Figura 5.2: Architettura del framework Torch

La versione del framework attuale è la 7, ma l'autore sta già lavorando ad una versione successiva, che dovrebbe essere quella definitiva (la 9) .

Torch7 ha oltre 50,000 download e una licenza libera BSD. È attualmente utilizzato da molte università tra cui la NYU e importanti aziende come IBM, il dipartimento di intelligenza artificiale di Facebook, Google DeepMind e altre[23].

5.1.1 Supporto CUDA

CUDA (Compute Unified Device Architecture) è l'architettura di elaborazione in parallelo di NVidia che permette netti aumenti delle prestazioni di computing grazie allo sfruttamento della potenza di calcolo delle GPU per operazioni "general purpose".

Torch offre un package chiamato `cunn` per usufruire di CUDA. Cunn è basato su un tensore chiamato "`torch.CudaTensor()`" che non è altro che un normale Tensor che utilizza la memoria della DRAM della GPU; tutte le operazioni definite per l'oggetto Tensor sono definite normalmente anche per il CudaTensor, il quale astrae completamente dall'utilizzo della GPU, offrendo un'interfaccia semplice e permettendo di utilizzare gli stessi script che si sono usati per l'elaborazione CPU. L'unica modifica da apportare è cambiare il tipo di tensore.

```
tf = torch.FloatTensor(4,100,100) -- CPU's DRAM
tc = tf.cuda() -- GPU's DRAM
tc:mul(3) -- eseguito dalla GPU
res = tc:float() -- res è istanziato nella DRAM della CPL
```

Figura 5.3 : Utilizzo di tensore con CUDA

5.2 Installazione

L'installazione è abbastanza semplice, siccome i mantainer del progetto forniscono due script principali: il primo installa le dipendenze e le librerie che LuajIT e Torch richiedono; il secondo installa LuajIT , Luarocks e utilizza poi Luarocks per installare Torch e i package più comuni e utili.

Questi script sono testati su Ubuntu ≥ 12 (si possono anche installare su altre distribuzioni se li si scarica e li si compila autonomamente) e OS X ≥ 10.8 .

Una volta installato, Torch7 prevede una shell interattiva dove poter testare comandi e script in maniera agevole.

```

Terminal

File Edit View Search Terminal Help

th> t = torch.Tensor(4,2)
[0.0971s]

th> t:fill(3)
3 3
3 3
3 3
3 3
[torch.DoubleTensor of dimension 4x2]
[0.0002s]

```

Figura 5.4: Torch7: Shell interattiva

Se si desidera una visualizzazione grafica Torch7 fornisce due strumenti utili:

1. ‘qlua’

il quale esegue Torch caricando il modulo relativo al famoso toolkit per la visualizzazione grafica QT [31]. Richiede di installare il package qlua.

2. ‘luajit -l gfx.go’

Il quale una volta lanciato fornisce una visualizzazione grafica basata su javascript + html visualizzabile dal browser all’indirizzo localhost:8000 (Figura 5.5).

Richiede l’installazione del package gfx.js ➔ luarocks install gfx.js

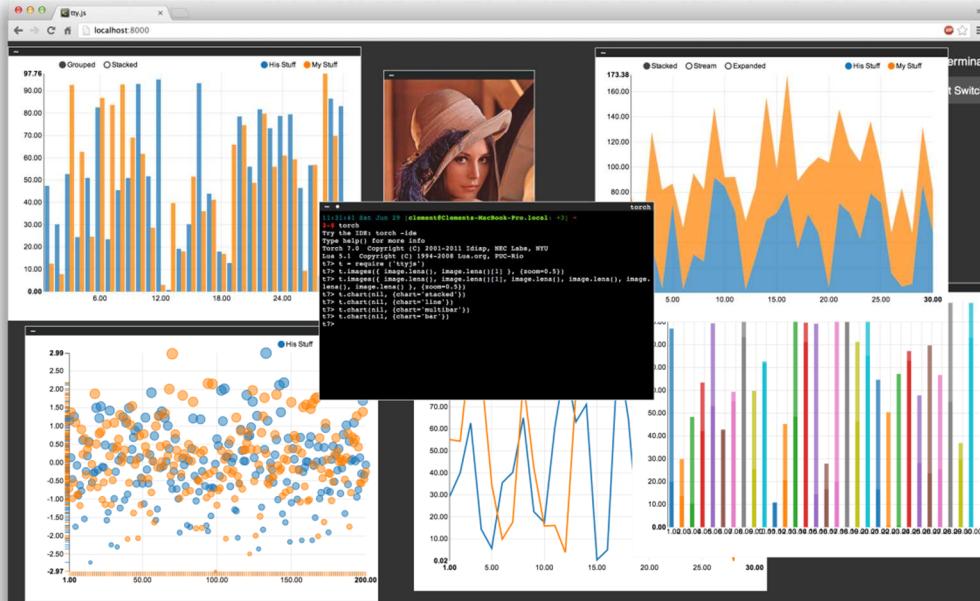


Figura 5.5: Visualizzazione tramite browser [24]

5.3 Utilizzo

5.3.1 Premessa

Ho utilizzato Torch7 con lo scopo di imparare ad usare i suoi strumenti per l'apprendimento supervisionato (supervised learning).

Quindi mi accingo ora a illustrare alcuni fondamentali passi del suddetto apprendimento, reperibili anche su un ottimo tutorial di C. Farabet [33], uno dei mantainer del progetto Torch. Tutti i vari step possono essere scritti in unico file Lua oppure in file separati e richiamati in un unico file col metodo '`require <nomefile>`' che serve a reperire e includere librerie.

5.3.2 Dataset

Al fine di allenare una rete abbiamo bisogno, prima di tutto, di un dataset su cui allenarla. Tale dataset, come spiegato nel capitolo 3, deve avere un numero di esempi equilibrato per ogni categoria e possibilmente un numero extra di esempi per il testing della rete.

Si può scaricare un dataset dalla sua pagina web, tramite browser o tramite un web crawler (arrampicatore del web) da terminale, come '`wget`' .

Ad ogni modo quest'operazione si può includere direttamente dentro gli script in Lua per Torch7, cosa che è infatti presente nei vari tutorial. Gli script utlizzano il comando Lua: `os.execute('comando')`, che permette di eseguire un comando dalla shell del sistema operativo.

Una volta scaricato il dataset, lo si può manipolare a piacimento. Ogni dataset ha però le proprie caratteristiche, come la risoluzione delle immagini che esso contiene e il numero degli esempi.

Per poterli utilizzare bisogna caricare le immagini e rielaborarle per farle diventare compatibili con la rappresentazione di Torch7.

Generalmente si può fare così:

- si creano due tensori, uno per le immagini e uno per le label.
- Si cicla su tutte le immagini presenti nella directory, le si carica in un tensore e gli assegna una label. Per assegnare correttamente una label si associa un valore ad ogni categoria e si assegna - per ogni esempio presente nel tensore delle immagini un valore corrispondente nel tensore delle label.

Esempio:

Le immagini sono dati a 3-dimensioni: [spazio-colore,altezza,larghezza].

Generalmente si aggiunge una dimensione che è l'indice per indicizzare tutte le N immagini che vogliamo caricare, quindi si crea un tensore di questo tipo :

t = torch.Tensor(N,3,400,400)

dove N sta per il numero di immagini che dobbiamo utilizzare. Si crea così un tensore in cui ad ogni indice corrisponde un'immagine 3x400x400.

Supponiamo ora di associare queste etichette: categoria cane= '1' categoria albero ='2'.

Allora si dovranno avere le associazioni tra label e immagini come nella figura d'esempio 5.6.

```
tensore_immagini[4] = 'cane8.jpg'    -- tra [] si indica l'indice come in un normale array
tensore_immagini[56] = 'albero7.jpg'
tensore_label[4] = 1                  -- all'indice corrispondente vi è la giusta Label!
tensore_label[56] = 2
```

Figura 5.6: corrispondenza label e immagini

Infine si fa un preprocessing del dataset equalizzando tutte le immagini del tensore sia localmente che globalmente, a seconda di quale dei due approcci abbia effetti migliori su un determinato dataset.

Dai miei test ho riscontrato che solitamente conviene trasformare le immagini da RGB in YUV per separare il canale della luminanza Y da quelli dei colori U,V. Dopodiché si normalizza Y localmente (cioè lo si normalizza per ogni esempio) e U,V globalmente.

Per normalizzare localmente si utilizza un modulo che si può includere nel modello della rete, mentre per normalizzare globalmente invece si fa "a mano": si ricava la media (Tensor:mean()) e la deviazione standard (Tensor:std()) poi globalmente si sottrae la media e si divide per la deviazione std.

Gli esempi di entrambi gli approcci sono mostrati in figura 5.7 e 5.8

```

38 -- Convertiamo tutte le immagini in YUV
39 print '==> preprocessing data: colorspace RGB -> YUV'
40 for i = 1,trainData:size() do
41   trainData.data[i] = image.rgb2yuv(trainData.data[i])
42 end
43 -- Vettore col nome dei canali, per comodità
44 channels = {'y','u','v'}
45 --normalizziamo ogni canale globalmente e salviamo il parametro mean e std
46 --relativi a media e deviazione std, perchè sono utili al training
47 mean = {}
48 std = {}
49 for i,channel in ipairs(channels) do
50   mean[i] = trainData.data[{},i,{},{}]:mean()
51   std[i] = trainData.data[{},i,{},{}]:std()
52   trainData.data[{},i,{},{}]:add(-mean[i])
53   trainData.data[{},i,{},{}]:div(std[i])
54 end

```

Figura 5.7: Esempio di normalizzazione globale

```

56 -- normalizzazione locale
57 -- Define un kernel per la normalizzazione
58 kernel = image.gaussian1D(13)
59 -- Definiamo l'operatore per la normalizzazione, è un modulo di nn,
60 --che si può quindi inserire direttamente nella CNN
61 normalization = nn.SpatialContrastiveNormalization(1, neighborhood, 1):float()
62 -- Normalize all channels locally:
63 for c in ipairs(channels) do
64   for i = 1,trainData:size() do
65     trainData.data[{},i,{},{}] = normalization:forward(trainData.data[{},i,{},{}])
66   end
67   for i = 1,testData:size() do
68     testData.data[{},i,{},{}] = normalization:forward(testData.data[{},i,{},{}])
69   end
70 end

```

Figura 5.8: Esempio di normalizzazione Locale

La sintassi per indicizzare righe e colonne è dei tensori è simile a quella di MatLab e può essere a prima vista un po' difficile , ma in realtà, grazie ad un veloce tutorial reperibile sempre sul sito del progetto, basta qualche esempio per poterla apprendere.

5.3.2.1 Il package Image

Se le immagini non sono tutte della stessa risoluzione si deve usare l'utilissimo package 'image' che consente di tagliare, scalare, aggiungere del padding, ecc alle immagini.

Fra le varie funzioni vi è anche l'importante SpatialPyramid che consente di avere una serie di immagini di risoluzioni diverse, per permettere alla rete di riconoscere le features anche se queste sono troppo grandi per le dimensioni dei receptive fields.

Ad esempio io ho utilizzato sempre reti che avevano bisogno di un input piccolo di 32x32 ma ovviamente la maggioranza delle immagini deve essere scalata o ridotta.

Se nonostante la risoluzione sia adatta la rete, il soggetto della foto è ripreso in posizioni troppo ravvicinate la rete potrebbe non essere in grado di riconoscere alcune caratteristiche.

Ad esempio una faccia in primo piano è decisamente più grande di una in una foto a distanza di 4,5 metri. In questo caso si fa quella che viene chiamata Spatial Pyramid dell'immagine; ovvero si prendono diverse risoluzioni della stessa immagine e le si mettono insieme in unico input per poter dare alla rete l'opportunità di estrarre le features correttamente.

5.3.3 Definizione di una CNN

Per costruire una rete neurale è sufficiente qualche riga di codice. Il package di riferimento in questo caso è nn e per ottenere un componente di una rete basta utilizzare i suoi metodi.

Vediamo alcuni esempi.

```
72 --[[ prima di tutto utilizziamo require, una funzione Lua per
73     |     reperire e caricare librerie. Carichiamo il package nn
74     |     (che sta per Neural Network) per Torch]]
75 require 'nn'
76 --[[semplice rete neurale a 2 strati, con
77 numero di hidden Layer = nhiddens ]]
78 model = nn.Sequential()
79 model:add(nn.Reshape(ninputs))
80 model:add(nn.Linear(ninputs,nhiddens))
81 model:add(nn.ReLU()) --si può sostituirlo con nn.Tanh()
82 model:add(nn.Linear(nhiddens,noutputs))
```

Figura 5.9: costruire un tradizionale MLP

Come spiegato più volte nel capitolo precedente, una CNN è tradizionalmente costituita da vari layer e terminata con un MLP. Quindi il codice per il classificatore MLP finale è lo stesso mentre bisogna aggiungere il codice per costruire gli strati della CNN.

Ne vediamo un esempio nella figura 5.10, dove si definiscono prima alcuni parametri del problema importanti come il numero delle classi, gli input, la misura dei kernel di convoluzione, di quelli di pooling e il numero di output di ogni livello.

```

87 --definiamo alcuni parametri
88 num_classi = 10 --problema da 10 classi
89 nfeats = 3 --[[numero feature map in ingresso sono 3 a causa
90 | | | | dello spazio colore RGB o YUV o altri]]
91 nstates = {16,64,128} --n. output di ogni livello
92 filtsize = 5
93 poolsize = 2
94 normkernel = image.gaussian1D(7)
95
96 --prima otteniamo un modello sequenziale dove aggiungere i componenti
97 model = nn.Sequential()
98 -- strato 1 : filter bank -> squashing -> max pooling -> normalization
99 model:add(nn.SpatialConvolutionMM(nfeats, nstates[1], filtsize, filtsize))
100 model:add(nn.ReLU())
101 model:add(nn.SpatialMaxPooling(poolsize,poolsize,poolsize,poolsize))
102 model:add(nn.SpatialSubtractiveNormalization(nstates[1], normkernel))
103
104 -- strato 2 : filter bank -> squashing -> max pooling -> normalization
105 model:add(nn.SpatialConvolutionMM(nstates[1], nstates[2], filtsize, filtsize))
106 model:add(nn.ReLU())
107 model:add(nn.SpatialMaxPooling(poolsize,poolsize,poolsize,poolsize))
108 model:add(nn.SpatialSubtractiveNormalization(nstates[2], normkernel))
109
110 -- strato 3 : standard 2-layer neural network
111 model:add(nn.Reshape(nstates[2]*filtsize*filtsize))
112 model:add(nn.Linear(nstates[2]*filtsize*filtsize, nstates[3]))--128 hidden units
113 model:add(nn.ReLU())
114 model:add(nn.Linear(nstates[3], num_classi))

```

Figura 5.10: Torch: Modello di una CNN

Notiamo anche che è stato aggiunto un modulo per la normalizzazione: 'SpatialSubtractiveNormalization'.

Se inserito all'inizio può evitare di dovere eseguire prima la normalizzazione però ovviamente ha il contro di aumentare il tempo di elaborazione. Utilizzare una serie immagini già normalizzate è più rapido rispetto a doverle normalizzare ad ogni ciclo.

Se si include la normalizzazione negli strati successivi (come in figura si ha invece l'interessante effetto di aumentare la "competizione" fra le features, facendo risaltare quelle più importanti.

Inoltre il modulo di Max pooling è ovviamente intercambiabile con uno di LP-pooling ottenibile così:

```
nn.SpatialLPPooling(numInput, normP, poolsize, poolsize, poolsize)
```

Per eventuali dettagli si rimanda alla documentazione sul sito del progetto.

5.3.3.1 Note importanti

Alcuni esempi non erano aggiornati e ho notato che utilizzavano componenti della rete che potevano provocare qualche ritardo, o meglio non essere ottimizzati.

Difatti, alla luce di quello che è stato detto dai ricercatori sui collegamenti tra le feature maps e i kernel dei convolutional layers (Capitolo 4), è stato aggiunto dagli autori un nuovo modulo che utilizza collegamenti random per migliorare le prestazioni.

Il modulo si istanzia sempre utilizzando nn:

```
nn.SpatialConvolutionMap(nn.tables.random(nfeats, noutputs, connections), filtsize, filtsize))
```

Dove nn.tables.random garantisce un numero di connessioni casuali, nfeats specifica il numero di feature maps in ingresso e connections è il numero di connessioni per ogni unità.

5.3.4 Loss function

Ora che abbiamo creato un modello dobbiamo definire una funzione di costo da minimizzare che, come visto nel capitolo 3, sta alla base dell'apprendimento.

Con Torch si possono definire diverse funzioni di costo, a seconda del principio di apprendimento che vogliamo utilizzare. Come ad esempio lo scarto quadratico che però non è in generale una buona scelta. Una delle più utilizzate e che ho trovato più conveniente è

stata la Negative-Likelihood (Likelihood = funzione di verosimiglianza[31]) [30]. Possiamo prima di tutto trasformare l'output del classificatore in uscita in probabilità logaritmica aggiungendo un modulo chiamato LogSoftMax e poi minimizzare l'errore minimizzando la negative-likelihood. In Torch questo è semplicissimo, come si vede nella figura 5.11.

```

144 --*****Negative Likelihood*****
145 --si aggiunge il modulo di LogSoftMax
146 -- per ottenere probabilità normalizzate logaritmiche
147 model:add(nn.LogSoftMax())
148 --[[La loss function in questo prende in ingresso
149 il vettore delle log-probabilities e come obiettivo
150 la classe da predire e ne misura il discostamento]]
151
152 criterion = nn.ClassNLLCriterion()
153
154 --**** SCARTO QUADRATICO MEDIO (MSE in inglese)
155 -- prima aggiungiamo una fz non lineare
156 --|per restringere l'output
157 model:add(nn.Tanh())
158
159 criterion = nn.MSECriterion()
160 criterion.sizeAverage = false
161 --lo scarto quadratico richiede anche di cambiare
162 --tutto il vettore delle label, per maggiori info
163 --si veda la documentazione.
164 --Non è comunque una funzione consigliata
165 end

```

Figura 5.12: Definire una loss function

L'esempio non è esaustivo, per maggiori informazioni si veda la documentazione. Ad ogni modo ai fini della tesi, come loss function si può scegliere la negative likelihood e utilizzarla a scatola nera. Una profonda analisi di come opera non è negli obiettivi della tesi.

5.3.5 Training della rete

Torch permette di allenare la rete tramite diversi algoritmi di ottimizzazione basati sulla discesa del gradiente (si veda capitolo 3). Si definisce una funzione usata per il training e la si manda in esecuzione dopo i passi che abbiamo visto fino a ora.

Un'altra cosa importante da definire sono il learning rate, ovvero il parametro che stabilisce la velocità di learning della rete per ogni esempio (quindi la sensibilità ad ogni esempio) e il lotto di esempi su cui fare la back propagation ad ogni iterazione (batch).

Grazie al supporto delle closure di Lua, si possono cambiare metodi di ottimizzazione e il size del batch a runtime.

L'apprendimento stocastico (casuale, la rete viene aggiornata per ogni immagine) è molto importante per la convergenza del problema, quindi si hanno migliori risultati con un batch-size = 1 (puramente stocastico) o un numero comunque basso, come ho potuto testare varie volte.

Obiettivo del training è quello di diminuire il discostamento tra la classe predetta e quella desiderata. Questo parametro lo misura la loss function, quindi parte del training è trovare il minimo della suddetta loss function.

Per il training di MLP ci sono metodi di poche righe di codice (figura 5.13) mentre per le CNN l'intera procedura di training è un po' più complicata, ma ci sono svariati file di esempio che si possono utilizzare a scatola chiusa. Il training dopotutto, è uguale per tutte le CNN tradizionali.

```
require "nn"
mlp = nn.Sequential(); -- make a multi-layer perceptron
inputs = 2; outputs = 1; HUs = 20; -- parameters
mlp:add(nn.Linear(inputs, HUs))
mlp:add(nn.Tanh())
mlp:add(nn.Linear(HUs, outputs))
```

Figura 5.13: Esempio di Training tramite il modulo StochasticGradient

Fornisco infine un caso di esempio (Figura 5.14) per applicare l'algoritmo della discesa del gradiente, che ho utilizzato io, reperibile sui tutorial. I commenti nel codice cercano di

spiegare i vari passaggi ma possono risultare non chiari se non si guarda lo script completo, per il quale rimando ai tutorial sul sito di Torch.

Una volta definita una funzione di training basterà fare un loop e invocarla per allenare la rete.

```
148      -- create closure to evaluate f(X) and df/dX
149      local feval = function(x)
150          -- se non vi sono già
151          -- si prendono i nuovi parametri di valutazione
152          if x ~= parameters then
153              parameters:copy(x)
154          end
155          -- si resettano i valori dei gradienti
156          gradParameters:zero()
157
158          -- f(x) è la media di tutti gli errori
159          local f = 0
160
161          -- cerchiamo una stima della fz di loss
162          -- e della sua derivata per ogni esempio
163          --[[inputs sono un numero=batch size
164          di immagini in ingresso ]]
165          for i = 1,#inputs do
166              local output = model:forward(inputs[i])
167              local err = criterion:forward(output, targets[i])
168              f = f + err
169
170              -- si stima df/dw per minimizzare la funzione d'errore
171              local df_do = criterion:backward(output, targets[i])
172              model:backward(inputs[i], df_do) -- back propagation
173
174              -- aggiorna la confusion matrix
175              confusion:add(output, targets[i])
176          end
177
178          -- normalizza gradienti e la f(X)
179          gradParameters:div(#inputs)
180          f = f/#inputs
181
182          -- return f and df/dX
183          return f,gradParameters
184      end
185
```

Figura 5.13: codice per stimare la discesa del gradiente

5.3.6 Testing della rete

Una cosa comune per chi utilizza Torch è testare la rete mentre la si allena. Questo solitamente è fatto su una parte più piccola del dataset utilizzata per la validation.

Per farlo si crea una funzione come per il training. In questa funzione si cicla sugli esempi memorizzati nei tensori e si da in ingresso alla rete un'immagine per volta.

Il metodo per fare elaborare l'immagine alla rete è:

```
output = model:forward(immagine_in_input)
```

dove 'model' è la rete.

Dopodiché si aggiorna la 'confusion matrix'[32] che è una matrice che indica per ogni classe del problema, il numero di esempi classificati correttamente e, per quelli errati, mostra con quale classe li si è confusi.

Una volta definita sia la funzione di testing che quella di training si può allenare e testare la rete a piacimento con un semplice while loop.

```
211 -- semplice Loop per allenare fino
212 -- a che non lo si interrompe
213 -- forzatamente
214 while true do
215   train(trainData)
216   test(testData)
217 end
218 --se vogliamo aggiungere una
219 --condizione di uscita
220 while true do
221   complete = train(trainData)
222   test(testData)
223   -- training 100% allora break
224   if complete == 1.0 then
225     break
226   end
227 end
228
```

Note:

Altre funzioni utili da citare sono quelle per scrivere e leggere file, ad esempio per salvare il training della rete e il caricamento della immagini.

Per salvare il modello della rete con i parametri già allenati, per poi riprendere l'allenamento in un secondo momento:

```
torch.save('nome_file.net',model), dove model è la nostra rete  
per caricarlo, invece:
```

```
model=torch.load('nome_file.net')
```

Similarmente per le immagini, si utilizza `image.load()` e `image.save()`. Sono supportati tutti i tipi di immagini.

5.4 Esempi applicativi

5.4.1 MNIST

MNIST è un dataset di cifre scritte a mano, disponibile sul sito di Yann LeCun [33], formato da un training set di 60'000 esempi e un test set di 10'000. Le immagini sono già state normalizzate e sono formattate in modo da avere la cifra esattamente al centro dell'immagine, e per questo richiedono un pre-processing praticamente nullo.

Per addestrare una rete a riconoscere le immagini di questo dataset basta scrivere un file (chiamiamolo ad esempio run.lua), contenente l'insieme delle funzioni viste nel capitolo precedente.

Poi da terminale si lancia ' th run.lua '

Il quale eseguirà lo script che addestrerà la rete dividendo il training in *epoch*e. Un'epoca è un'iterazione completa su tutto il dataset.

Un esempio di output da terminale è riportato in figura 5.14. Si nota la confusion matrix, l'epoca di training e le percentuali di esempi classificati correttamente, sia per il training che per il testing.

```
<trainers> time to learn 1 sample = 24.487635493279ms===== ETA: 324ms | Step: 36ms
confusionMatrix:
[[ 190      0      0      0      0      0      1      0      0      0] 99.476% [class: 0]
[ 0     217      1      1      0      0      0      1      0      0] 98.636% [class: 1]
[ 0      0    197      0      0      0      1      0      0      0] 99.495% [class: 2]
[ 0      0      0    189      0      0      1      0      0      1] 98.953% [class: 3]
[ 0      0      0      0    212      0      1      0      0      0] 99.065% [class: 4]
[ 0      0      0      0      0   180      0      0      0      0] 100.000% [class: 5]
[ 1      0      0      0      0      0   199      0      0      0] 99.500% [class: 6]
[ 0      0      0      0      0      0   224      0      0      0] 100.000% [class: 7]
[ 0      0      0      0      0      0      0   171      1      0] 99.419% [class: 8]
[ 2      0      0      1      0      0      0      0      0   207]] 98.571% [class: 9]
+ average row correct: 99.311608672142%
+ average rowUcol correct (VOC measure): 98.63573372364%
+ global correct: 99.3%
<trainers> saving network to /home/synchro/Dropbox/Università/Tesi/github/demos/train-a-digit-classifier/logs/mnist.net
<trainers> on testing Set:
<trainers> time to test 1 sample = 20.104405999184ms===== ETA: 155ms | Step: 17ms
confusionMatrix:
[[ 190      0      0      0      0      0      1      0      0      0] 99.476% [class: 0]
[ 0     217      1      1      0      0      0      1      0      0] 98.636% [class: 1]
[ 0      0    197      0      0      0      1      0      0      0] 99.495% [class: 2]
[ 0      0      0    189      0      0      1      0      0      1] 98.953% [class: 3]
[ 0      0      0      0    212      0      1      0      0      0] 99.065% [class: 4]
[ 0      0      0      0      0   180      0      0      0      0] 100.000% [class: 5]
[ 1      0      0      0      0      0   199      0      0      0] 99.500% [class: 6]
[ 0      0      0      0      0      0   224      0      0      0] 100.000% [class: 7]
[ 0      0      0      0      0      0      0   171      1      0] 99.419% [class: 8]
[ 2      0      0      1      0      0      0      0      0   207]] 98.571% [class: 9]
+ average row correct: 99.311608672142%
+ average rowUcol correct (VOC measure): 98.63573372364%
+ global correct: 99.3%
```

Figura 5.14: Confusion matrix e percentuali di accuracy sul training e testing, MNIST

Ho allenato la rete fino al 99,3% di precisione sul dataset di testing, dopodiché volendo avere un risultato visivo più concreto ho scritto del codice che permetteva di visualizzare, in una finestra del browser, la previsione della rete per ogni immagine di test.

Come è possibile osservare nella figura 5.15, la rete è effettivamente molto accurata.

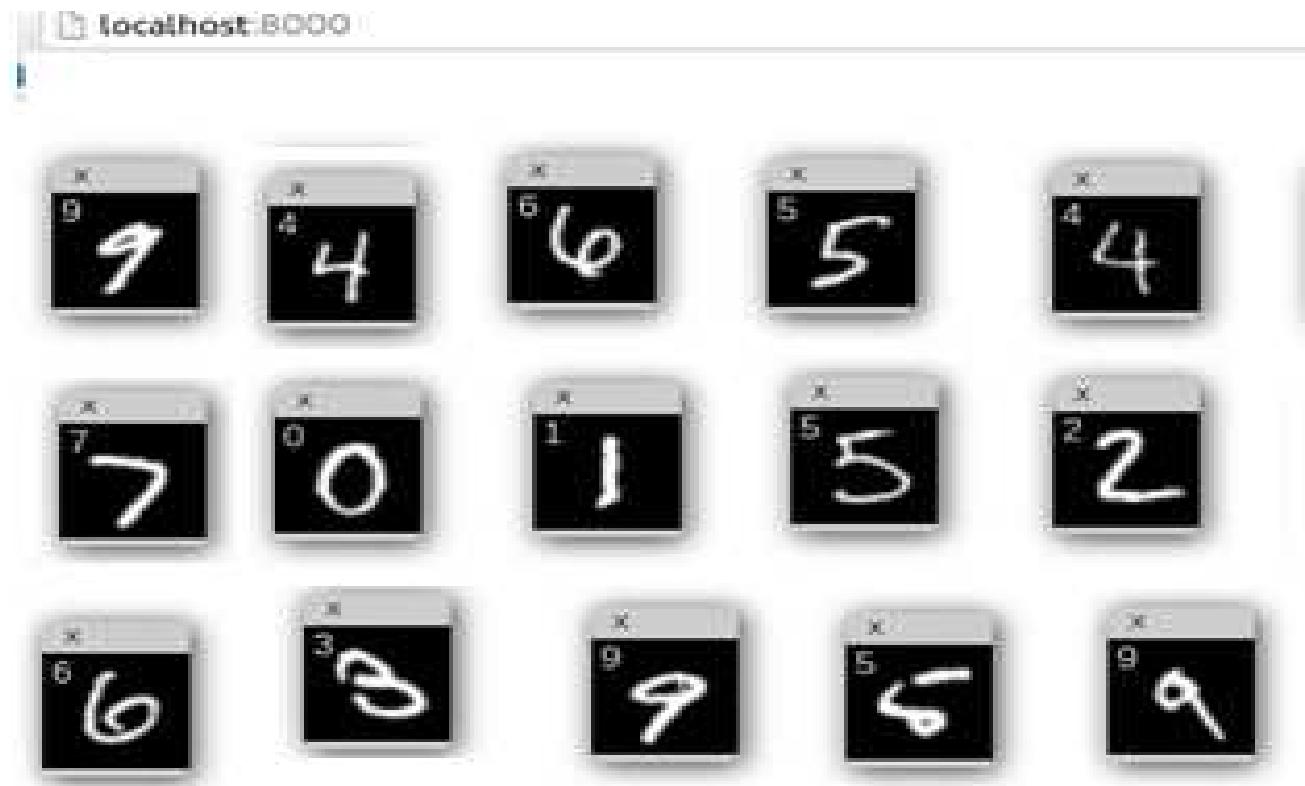


Figura 5.15: Predizione delle cifre. Nell'angolo in alto a sinistra il numero predetto.

5.4.2 CIFAR10

Il dataset CIFAR10 è decisamente più impegnativo in quanto contiene immagini a colori di 10 classi di oggetti e animali del mondo reale: aeroplani, automobili, uccelli, cani, cavalli, renne, rane, navi e camion.

Vi sono 60000 immagini a colori 32x32 divisi nelle suddette 10 classi, 6000 immagini per classe. Le immagini sono divise in 50000 per il training e 10000 per il testing.

Più precisamente vi sono 5 batch di training e 1 di testing.

Quello di testing contiene esattamente 1000 immagini per ogni classe, mentre alcuni batch di training contengono più immagini di una classe rispetto ad un'altra, ma nell'insieme contengono esattamente 5mila immagini per ogni classe.

Gli oggetti di ogni classe sono estremamente variabili, ad esempio ci sono tantissime specie di uccello, grandi e piccole. Non solo, gli uccelli sono anche in svariate pose diverse e a volte sono mostrate solo alcune parti del loro corpo. Lo stesso problema è anche peggiore per i cani e i gatti che vengono mostrati in tutte le pose immaginabili, a volte la testa non si vede e a volte invece si vede solo quella. Tutto questo lo rende un dataset piuttosto difficile per un sistema automatico. Il record assoluto nell'accuratezza nel riconoscimento risale al 2011 ed è del 94% [34].

Questo dataset si presta abbastanza bene agli sviluppi futuri della tesi, riguardante il sistema di assistenza alla deambulazione per non vedenti, poiché contiene molti oggetti che possono essere presenti nelle vie di una città, un parco, etc.

Come per MNIST, ci basta un comando per lanciare lo script che addestrerà la rete.

Utilizzando la libreria grafica ‘imgraph’ possiamo anche visualizzare dei grafici che indicano lo stato del training e del testing, e aggiornarli ad ogni epoca (Figura 5.17) .

Il modello della rete è quello di una CNN composta da 2 strati identici e un MLP finale. I due strati sono composti da *Convolutional layer- Tangente Iperbolica - Max pooling*.

Facendo un addestramento di 1 ora (3-4 epocha) raggiungiamo già un buon 62%, con picchi di quasi l'80% in alcune categorie, mentre rimangono basse le percentuali delle classi sopra presenti nel dataset.

Per avere un'ottimizzazione completa e completare il training servono 20 epocha.

Completato il training e, quindi passate 7 ore, la situazione è però quella in figura 5.16.

Possiamo notare alcune cose dalla confusion matrix. Ad esempio nella 4° colonna possiamo vedere i numeri di esempi classificati in modo errato come gatti, per ogni classe sull'asse Y a destra. Osserviamo che i cani sono stati classificati come gatti per ben 218 volte.

```

[[ 0 0 5000 0 0 0 0 0 0 0] 100.000% [class: bird]
[[ 0 0 0 5000 0 0 0 0 0 0] 100.000% [class: cat]
[[ 0 0 0 0 5000 0 0 0 0 0] 100.000% [class: deer]
[[ 0 0 0 0 0 5000 0 0 0 0] 100.000% [class: dog]
[[ 0 0 0 0 0 0 5000 0 0 0] 100.000% [class: frog]
[[ 0 0 0 0 0 0 0 5000 0 0] 100.000% [class: horse]
[[ 0 0 0 0 0 0 0 0 5000 0] 100.000% [class: ship]
[[ 0 0 0 0 0 0 0 0 0 5000]] 100.000% [class: truck]
+ average row correct: 100%
+ average rowUcol correct (VOC measure): 100%
+ global correct: 100%
<trainer> saving network to /home/synchro/Dropbox/Università/Tesi/github/demos/train-on-cifar/logs/cifar.net
<trainer> on testing Set:
[===== 10000/10000 =====>] ETA: 0ms | Step: 6ms
<trainer> time to test 1 sample = 6.9102224826813ms
ConfusionMatrix:
[[ 706 37 44 28 16 15 10 20 89 35] 70.600% [class: airplane]
[[ 37 784 9 16 4 9 11 6 41 83] 78.400% [class: automobile]
[[ 71 15 488 92 78 79 83 54 22 18] 48.800% [class: bird]
[[ 38 9 78 496 54 170 76 34 23 22] 49.600% [class: cat]
[[ 31 3 62 71 591 48 77 91 17 9] 59.100% [class: deer]
[[ 15 7 64 218 48 532 36 62 13 5] 53.200% [class: dog]
[[ 8 6 48 77 31 26 786 8 5 5] 78.600% [class: frog]
[[ 17 5 31 57 78 68 9 707 7 21] 70.700% [class: horse]
[[ 67 56 11 15 16 7 5 9 776 38] 77.600% [class: ship]
[[ 36 86 8 22 9 4 7 20 43 765]] 76.500% [class: truck]
+ average row correct: 66.310000121593%
+ average rowUcol correct (VOC measure): 50.375524759293%
+ global correct: 66.31%

```

Figura 5.16: Training e testing: risultati dopo 7 ore. Si notano le categorie problematiche bird, cat e dog.

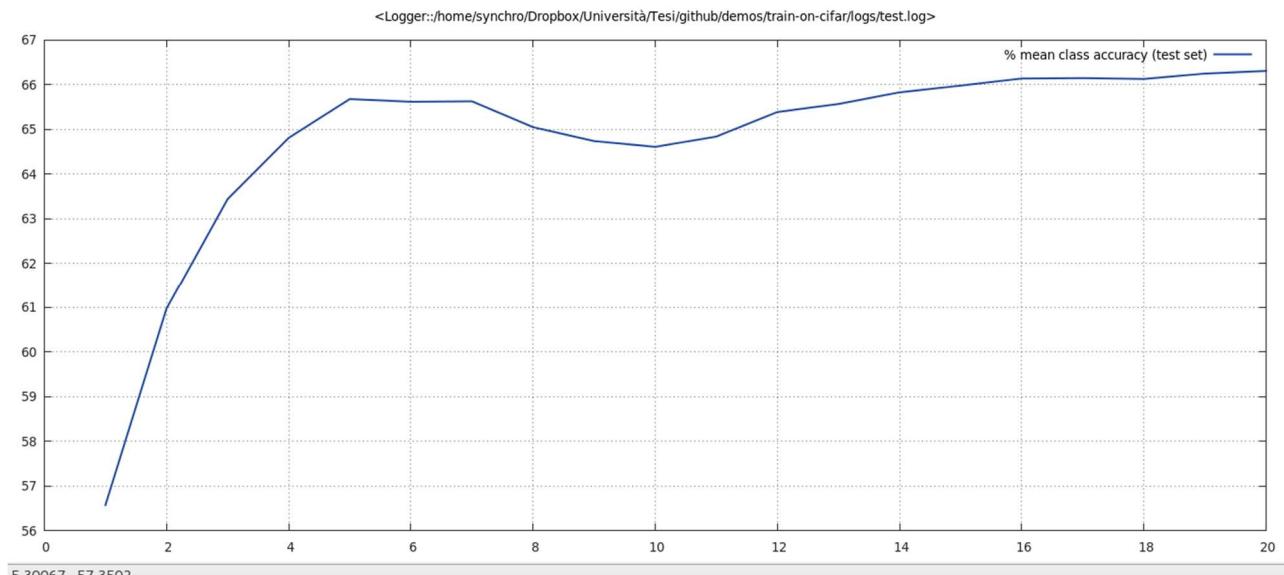


Figura 5.17: curva di apprendimento sul dataset di testing. Si stabilizza sul 66,5% di risposte corrette

È possibile osservare che i risultati sono rimasti all'incirca stabili dalla 3-4° epoca in poi e le categorie più difficili sono sempre quelle di cane, gatto e uccello.

Alla luce delle ultime ricerche, ho provato a sostituire la tangente iperbolica con una RELU e i risultati sono nettamente migliorati.

Dopo solo 4 epochhe, quindi 1 ora, si sono raggiunte percentuali del 67,57%, mai raggiunti tramite la tangente iperbolica (Figura 5.18).

In figura 5.19 invece, possiamo osservare il dato interessante che l'utilizzo della ReLU ha nettamente migliorato i risultati nella classificazione degli uccelli, con un margine del 20%.

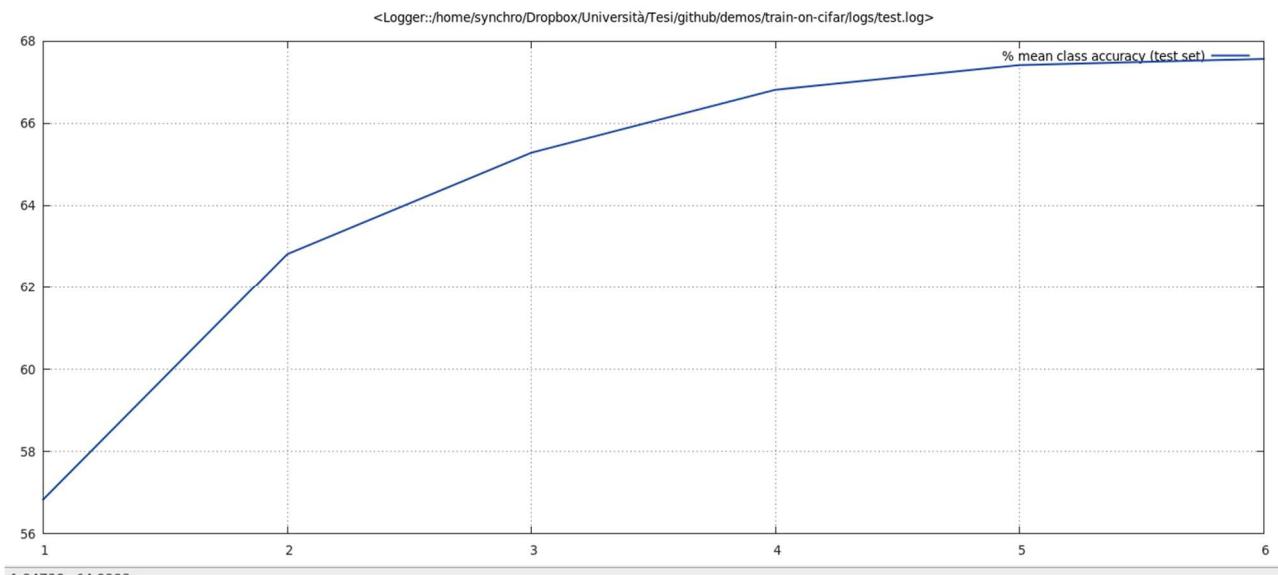


Figura 5.18: Risultati dopo appena 60 minuti, utilizzando la ReLU anziché la tangente iperbolica.

```

70.000% [class: airplane]
82.600% [class: automobile]
67.100% [class: bird]
49.800% [class: cat]
79.600% [class: deer]
40.100% [class: dog]
70.900% [class: frog]
72.100% [class: horse]
72.300% [class: ship]
71.200% [class: truck]

```

Figura 5.19: Percentuali di accuracy per ogni classe

Le percentuali migliorano ulteriormente se utilizziamo una funzione di pooling a norma 2, e aggiungiamo alla fine dei 2 convolutional layer uno strato di normalizzazione.

Dopo sole 4 epochhe si passa la soglia del 70% e dopo le solite 20 epochhe si riesce a raggiungere un 73-74%: il miglioramento è quindi notevole. (figura 5.20-5.21)

```
[===== 10000/10000 =====] ETA: 0ms | Step: 8ms
<trainer> time to test 1 sample = 10.493553209305ms
ConfusionMatrix:
[[ 775   24   36   23   15   12    7   16   63   29] 77.500% [class: airplane]
[ 24  842    4    9    3    3    6    6  28   75] 84.200% [class: automobile]
[ 64    4  611   79   83   59   52   27   12   9] 61.100% [class: bird]
[ 25    8   51  583   43  171   60   21   19   19] 58.300% [class: cat]
[ 18    4   48   50  701   48   50   62   11   8] 70.100% [class: deer]
[  9    3   49  207   47  584   30   59   6   6] 58.400% [class: dog]
[  3    2   38   61   32   34  811   10   6   3] 81.100% [class: frog]
[ 14    0   27   47   56   63    8   766   4   15] 76.600% [class: horse]
[ 72   24   14   11   11    7    4    7  806   44] 80.600% [class: ship]
[ 33   56    8   12   11    7    9   12   33  819]] 81.900% [class: truck]
+ average row correct: 72.979999184608%
+ average rowcol correct (VOC measure): 58.347061574459%
+ global correct: 72.98%
```

Figura 5.20: Confusion Matrix. Miglioramenti notevoli su tutte le classi.

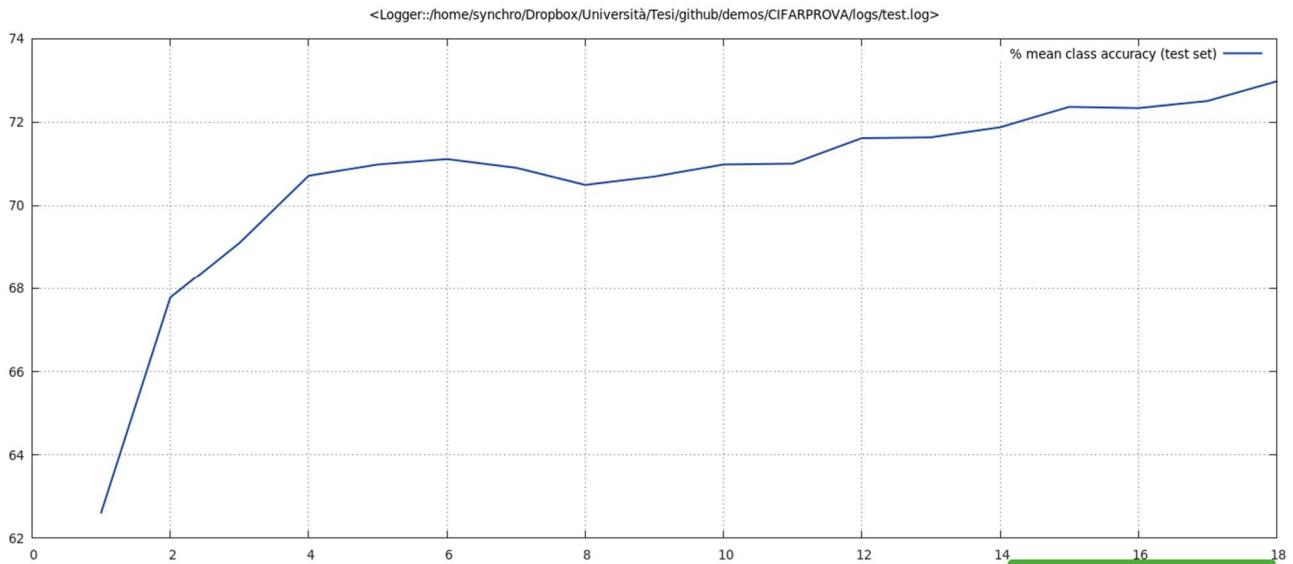


Figura 5.21: Risultati di test utilizzando anche uno strato di normalizzazione.

Questo perché lo strato di normalizzazione locale favorisce una sorta di ‘competizione tra le features’ , quindi la rete riesce ad estrarre caratteristiche più peculiari e utili ai fini della classificazione.

Anche in questo caso ho scritto del codice per veder nel browser come venivano classificate le immagini. In figura 5.22 sono mostrate esempi di classificazioni corrette con tanto di tempo di testing in millisecondi, mentre in 5.23 ci sono esempi di classificazioni errate. Si vede come la rete faccia confusione tra cani, gatti e uccelli.

I risultati raggiunti sono molto interessanti considerando che per il dataset in questione è sufficiente giungere ad un 77% per essere pubblicati nella classifica mondiale.

Ad ogni modo tutti gli approcci che sono in quella classifica applicano accorgimenti particolari e studiati, come un particolare pre-processing o una divisione delle categorie in cluster (sottogruppi) da valutare in modo separato.

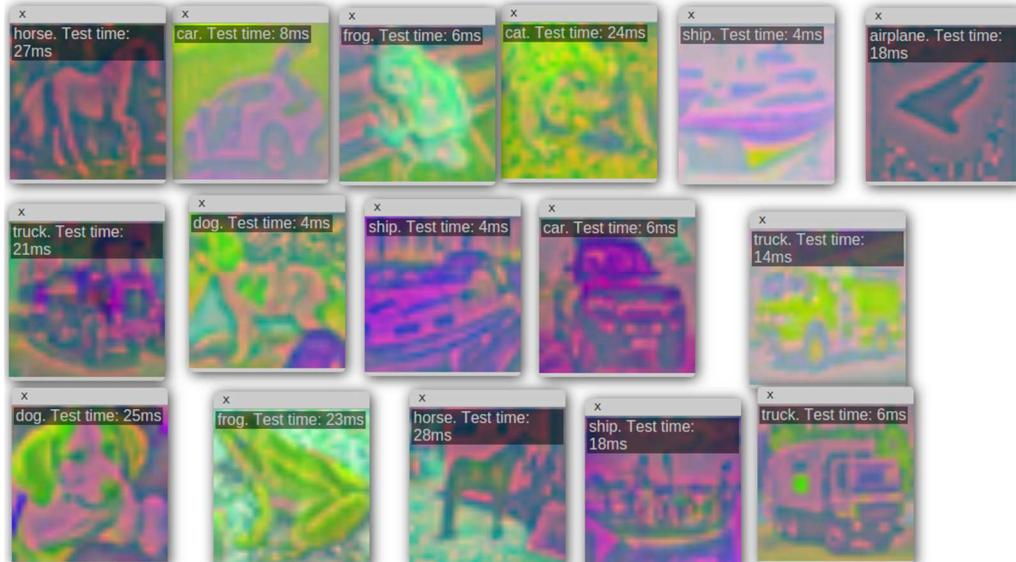


Figura 5.22: Esempi classificati correttamente con relativi tempi di testing. Le immagini hanno subito del pre-processing.

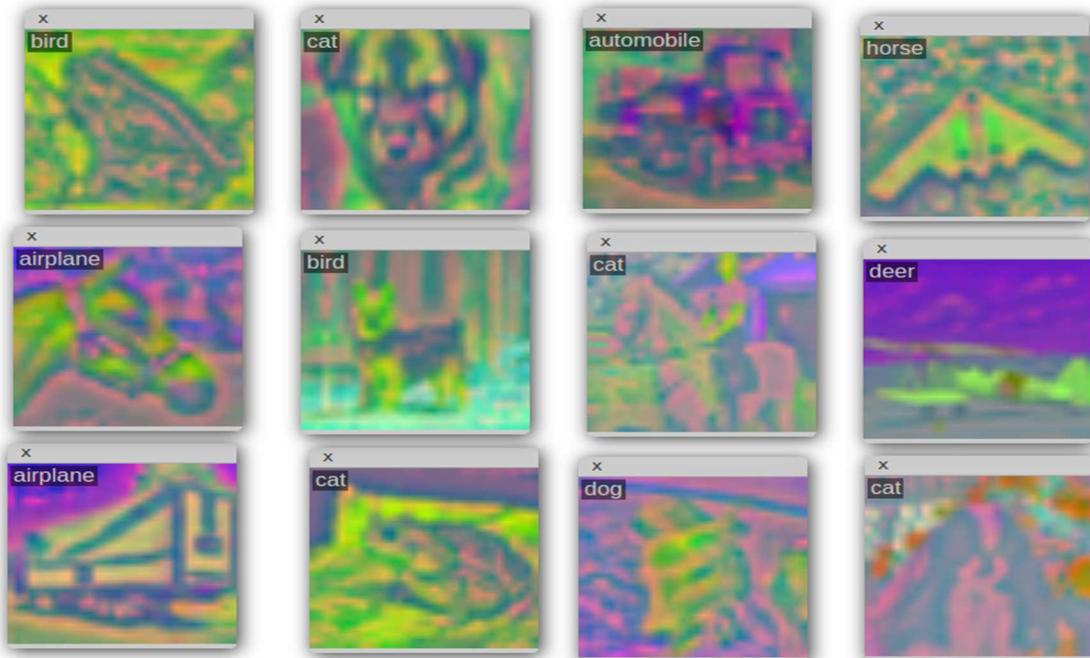


Figura 5.23: Immagini classificate in modo sbagliato

5.5 Casi di studio

Si volevano applicare gli algoritmi di Torch ad un sistema di visione 3D per l'assistenza alla deambulazione di persone non vedenti.

La scena ripresa in real-time da una persona che cammina per strada è decisamente più complessa: vi sono infatti molti oggetti insieme nella stessa scena ed ogni oggetto non è centrato nell'immagine e non è facilmente distinguibile dagli altri.

Un risultato efficace nella segmentazione e riconoscimento completo dell'immagine è stato riportato da Clement Farabet [33] utilizzando Torch e lo riporto in figura 5.24.



Figura 5.24: Parsing dell'immagine. Clement Farabet

Avendo però il vantaggio di una videocamera 3D, si può riconoscere e separare un ostacolo dallo sfondo calcolando la differenza di distanza tra i due. Quindi si segmenta l'oggetto e si ottiene un'immagine 2D simile a quelle usate per la classificazione. Queste immagini possono essere classificate senza troppe difficoltà tramite l'uso di CNN.

A questo scopo ho provato a creare un dataset scaricando da Internet una serie di immagini che potevano essere significative. Successivamente ho scritto un semplice script in bash per ciclare nella directory delle immagini e dare un nome significativo ad ogni immagine a seconda della categoria a cui appartiene.

Avere dei nomi significativi aiuta a caricare tutte le immagini in un tensore, tramite un semplice loop, come quello in figura 5.25.

```

230 --carichiamo i dati dal disco
231 --in questo caso 5000 immagini
232 local total = 5000
233 local imagesAll = torch.Tensor(total,3,32,32)
234 local labelsAll = torch.Tensor(total)
235
236 --Esempio con immagini 'png', ma si può cambiare decidendo da riga di comando il tipo
237 for i=1,#classes do
238   --load images
239   prefix=folders[i]..'/..folders[i] --prefix = apple/apple
240   local count = getNumber(folders[i]) -- retrieving total number of photos
241   for j=1,count do
242     imagesAll[{ {(i*count), (i*count+j)} }] = image.load(prefix..'.png') --apple/apple.1.png, apple/apple.2.png
243     labelsAll[{ {(i*count), (i*count+j)} }] = classes[i] --lega le foto caricate alla corretta label
244   --quindi ad ogni ciclo si caricano count immagini
245   --e gli ad ognuna si associa la label corretta
246   end
247 end

```

Figura 5.25 Un loop per creare un dataset labeled partendo dalle immagini salvate su disco.

Dove ‘getNumber’ è una funzione per contare quante foto sono presenti in una directory. Dopo aver caricato il mio dataset ho compiuto il training e il testing della rete giungendo, ovviamente per il ridotto numero di immagini, a scarse percentuali di correttezza.

In figura 5.26 vi sono alcune immagini del mio dataset prima e dopo la fase di preprocessing.



Figura 5.26: Elaborazione di un dataset arbitrario.

6. RISULTATI Sperimentali e confronto con il framework Caffe

6.1 Premessa

Come specificato nell'introduzione, scopo di questo elaborato era anche il confronto con il Framework Caffe, il quale si basa anch'esso sull'utilizzo delle CNN per il riconoscimento delle immagini e che sta avendo molto successo nella comunità scientifica. Tale framework è stato esaminato da Elisa Rimondi nel corso della sua tesi "Classificazione di immagini mediante il framework Caffe", e l'obiettivo comune era un confronto tra i due, mediante risultati sperimentali su alcuni dataset utilizzati da entrambi, come appunto MNIST e CIFAR10. Una volta scelto il framework più appropriato allo scopo dell'attività di ricerca in corso presso il DISI, lo si utilizzerà per il sistema per l'ausilio ai non vedenti già citato in precedenza.

Per quanto concerne il confronto tra Torch e Caffe, si riportano le specifiche dei sistemi di calcolo (differenti) utilizzati per la valutazione:

Torch7: I risultati sono stati ottenuti con una CPU Intel(R) Pentium(R) CPU P6200, 2.13GHz dual core, sistema operativo Ubuntu 14.04 nativo con 4Gb di RAM. Gli script venivano eseguiti utilizzando 2 thread.

Caffe: I risultati sono stati ottenuti con una macchina virtuale (Oracle VirtualBox) con Ubuntu 14.04, 4Gb di Ram e un processore single core. La VM era installata su un laptop con CPU: INTEL(R) Core (TM) i7-3610QM, 2.30GHz.

6.2 MNIST

Per addestrare la rete ho impiegato circa 40 minuti ottenendo un accuracy del 99.3%.

Per la singola immagine di esempio la rete la elaborava in 12ms per il training (quindi confronti, back propagation, ecc) e in soli 5ms per il testing (figura 6.1). Altri esempi si possono vedere nelle immagini relative al capitolo degli esempi applicativi.

Il training di Caffe ha richiesto 1 ora di training per arrivare anch'esso al 99%. Per la classificazione di un'immagine arbitraria ci mette un po' di più, 120ms (Figura 6.2). C'è da dire però, che – oltre all'essere eseguito su macchina virtuale - l'interfacciamento con le librerie di python richiede più tempo che quelle in Lua, questo potrebbe giustificare questa differenza. Il tipo di rete era simile, ma comunque non identico.

Per una comparazione più precisa sarebbe necessario utilizzare la stessa macchina, ma ad ogni modo i risultati sono stati similari, sia per accuracy che per tempistiche di training.

```
time to test 1 sample = 5.4291770458221ms:
```

Figura 6.1: Torch: tempo di classificazione di un digit del dataset MNIST, ≈5ms

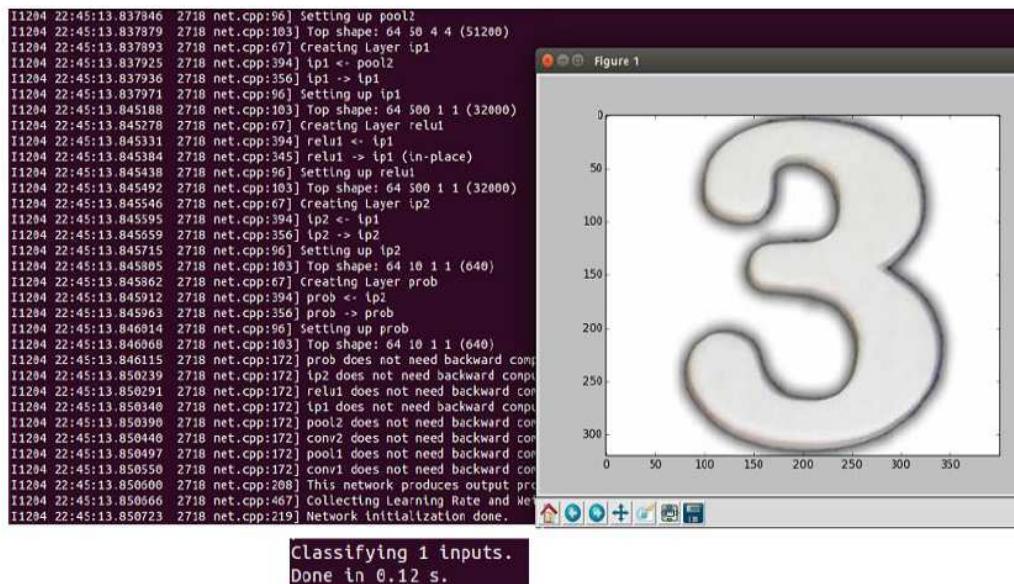


Figura 6.2: Caffe: classificazione di un digit in 120 ms.

6.3 CIFAR10

Anche per CIFAR10 si sono raggiunte percentuali di accuracy analoghe.

Nei miei esperimenti sono riuscito a raggiungere un 73% dopo circa 7 ore di addestramento della rete. Caffe ha raggiunto una performance del 75%, però solo dopo due ore di addestramento.

La rete con cui ho raggiunto i risultati migliori era quella che utilizzava una componente di normalizzazione, scelta condivisa anche dalla rete utilizzata con Caffe. I due metodi di normalizzazione utilizzati sono però diversi fra loro.

Per elaborare un'immagine di training Torch7 impiega mediamente 25-30 ms mentre per un'arbitraria immagine di testing si passa, a seconda dell'immagine, da estremi di 4ms anche ad 80ms, ma la media si aggira sui 15-20ms. In figura 6.3 vediamo un'immagine di esempio, che la rete (comprendente in questo caso anche uno strato di normalizzazione) classifica in 56ms. La stessa immagine impiega invece solo 17ms ad essere classificata da una rete senza layer di normalizzazione. Inoltre, ho scritto del codice per misurare la velocità di elaborazione anche nel caso di un preprocessing dell'immagine. I tempi di esecuzione rimangono comunque piuttosto ridotti, circa 24ms (figura 6.4).

Caffe dimostra anche qui di avere tempi di latenza più elevati figura (6.5), ma vale lo stesso discorso fatto per MNIST: sarebbe necessario testarlo sulla stessa macchina, cosa che faremo sicuramente presto.



```
th test_image.lua A3rossa.jpg logs/cifar.net
time to classify the sample: 56 ms
```

Figura 6.3: Torch: tempo di predizione della rete.

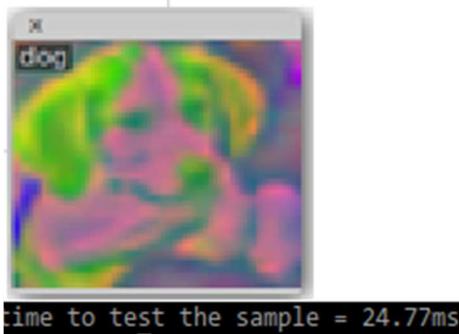


Figura 6.4: Torch: tempo di predizione della rete, compresa normalizzazione, scaling e cropping.

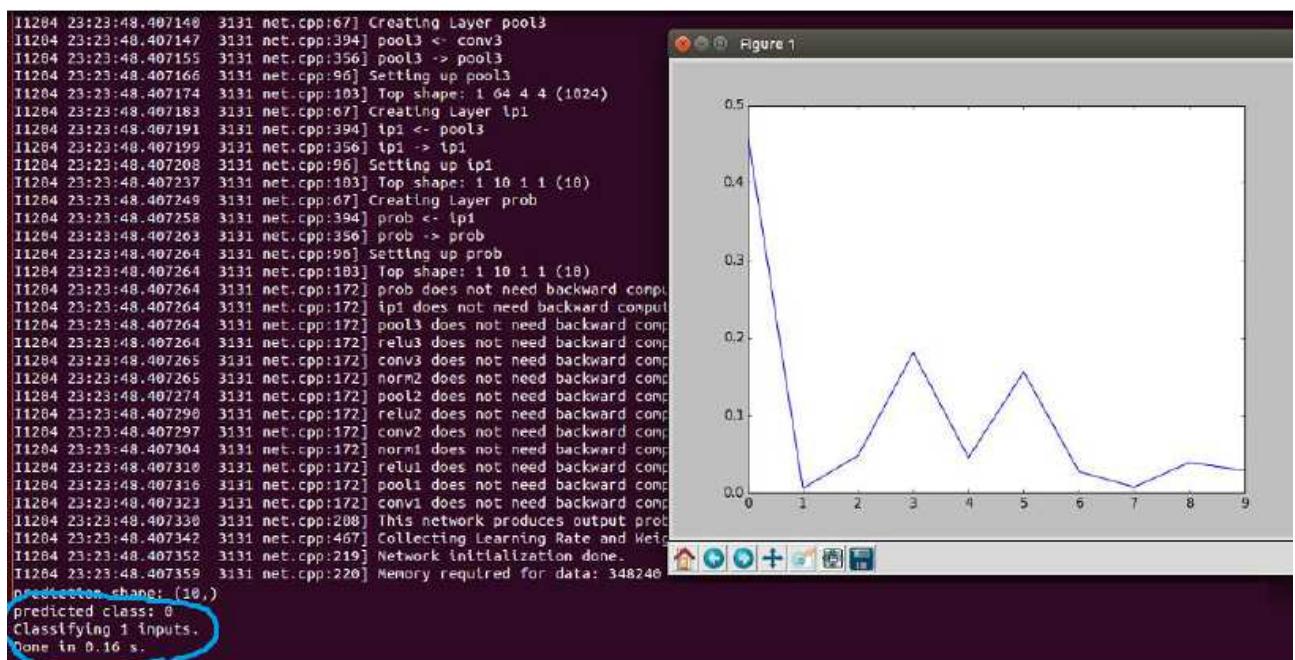


Figura 6.5: Caffe: classificazione dell'immagine di una automobile, evidenziato il tempo 160ms.

Un'ultima cosa da dire è che entrambi i framework supportano l'elaborazione mediante l'architettura CUDA, la quale potrebbe aumentare di un fattore notevole (anche 10 e più) la già interessante velocità di elaborazione delle immagini.

Non disponendo di un laptop con una GPU Nvidia non ho potuto sfruttare questa potenza di calcolo, e nemmeno la mia collega Elisa Rimondi ha potuto farlo a causa di problemi di installazione riscontrati con il proprio laptop.

7. CONCLUSIONI E SVILUPPI FUTURI

Durante questo lavoro mi sono occupato di studiare ed utilizzare il framework Torch7, il quale è risultato essere un ottimo strumento per il Deep Learning e in particolare per le Convolutional Neural Networks.

Mediante Torch7 sono state svolte numerose prove, sperimentando le diverse alternative proposte dalla documentazione e cercando di seguire gli approcci più promettenti citati nella letteratura consultata per affrontare il problema.

I risultati raggiunti possono considerarsi interessanti e confermano pienamente le aspettative avute sulle CNNs.

Scopo della tesi era anche quello del confronto fra Torch e Caffe, i due framework attualmente più promettenti nell'ambito delle CNNs. Alla luce degli esperimenti svolti sono giunto alle seguenti conclusioni.

Utilizzare le CNNs per il riconoscimento delle immagini, è sicuramente uno degli approcci più efficaci e sarà molto probabilmente oggetto di ricerca del DISI nei prossimi anni.

Nella comparazione tra i due framework emerge, a mio avviso, una preferenza per Torch7, almeno allo stato attuale delle cose.

Torch7 è risultato più facile da installare, ha una buona documentazione, un ottimo supporto da parte degli utenti ed è stato testato molte volte durante questi anni.

Caffe dal canto suo è un progetto decisamente giovane (Aprile 2014), con tutto ciò che questo comporta.

La mia collega Elisa Rimondi ha trovato molte difficoltà nell'installazione, una documentazione non ancora matura e un supporto minore rispetto a quello che io ho riscontrato con Torch7.

D'altra parte però, Caffe vanta il supporto di aziende con un know-how specifico del calibro di NVIDIA [36], e nel giro di pochi anni questo progetto potrebbe compiere grossi miglioramenti, diventando quindi un'alternativa ancor più interessante a Torch.

Ritengo infine che, anche grazie alla efficiente velocità di elaborazione dimostrata da Torch7, questo lavoro potrà avere diversi sviluppi, tra i quali, l'implementazione per la realizzazione del sistema di ausilio ai non vedenti, citato più volte nel corso dell'elaborato. A tal proposito sarebbe interessante trovare una maniera per classificare in modo accurato più input provenienti dalla stessa immagine, e magari arrivare alla realizzazione di un sistema di parsing completo.

Una ulteriore e interessante direzione da percorrere potrebbe essere quella di sfruttare il supporto a CUDA di Torch7, ed utilizzare una o due GPU in parallelo per l'addestramento di una CNN con più layer o due CNN in parallelo.

Ringraziamenti

Desidero ringraziare di cuore il Prof. Stefano Mattoccia per avermi dedicato il tempo e la pazienza di occuparsi della mia tesi, di farmi da relatore e per avermi fatto appassionare ad un campo molto interessante che ancora non conoscevo.

Un ringraziamento speciale va alla mia famiglia e a chiunque si impegni tutti i giorni ad essere presente, a cui dedico ogni mio sforzo e risultato.

Riferimenti

- [1] Fukushima, Kunihiko (1980). "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position".
- [2] R. Collobert, K. Kavukcuoglu and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In BigLearn, NIPS Workshop, 2011
- [3] Caffe: Convolutional Architecture for Fast Feature Embedding. Web Site. <http://caffe.berkeleyvision.org>
- [4] DAUGHMAN, J. "Complete discrete 2-D Gabor transform by neural networks for image analysis and compression". In: IEEE trans. Acoustic, Speech. 1988. p. 1169-1179.
- [5] BARBIERI, Alessandro; MESSINA, Alberto. "Strumenti per la Classificazione Automatica di Contenuti Audiovisivi". Politecnico di Torino. 2004
- [6] Mitchell, T. (1997). Machine Learning, McGraw Hill
- [7] McCulloch e Pitts(1943)"A logical calculus of the ideas immanent in nervous activity"
- [8] <http://www.toptal.com/machine-learning/an-introduction-to-deep-learning-from-perceptrons-to-deep-networks>
- [9] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato and Yann LeCun "What is the Best Multi-Stage Architecture for Object Recognition?"
- [10] D. E. Rumelhart, G. E. Hinton, R. J. Williams. Learning internal representations by error propagation, Parallel distributed processing: explorations in the microstructure of cognition, vol. 1, pp. 318-362, MIT Press Cambridge, MA, USA, 1986
- [11] Yann LeCunn, Soumith Chintala, Pierre Sermanet : Convolutional Neural Networks Applied to House Numbers Digit Classification
- [12] Deep Learning using Linear Support Vector Machine, Yichuan Tang
- [13] Andrew Saxe, Pang Wei Koh, Zhenghao Chen, Maneesh Bhand, Bipin Suresh, and Andrew Ng. On random weights and unsupervised feature learning. In International Conference on Machine Learning, 2011.
- [14] Krizhevsky, I. Sutskever, and G. Hinton. ImageNet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems, 25, 2012.
- [15] Eugenio Culurciello, Aysegul Dundar, Jonghoon Jin, Jordan Bates. An Analysis of the Connections Between Layers of Deep Neural Networks

- [16] www.teradeep.com/services
- [17] <http://yann.lecun.com/exdb/mnist/>
- [18] 12 May 2014: Ali Sharif Razavian, CVAP KTH, CNN Features off-the-shelf: an Astounding Baseline for Recognition
- [19] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. arxiv:1311.2524 [cs.CV]
- [20] VLAD, BoW, IFV, Hamming Embedding and BoB.
- [21] Mrazova, I.; Pihera, J.; Veleminska, J., "Can N-dimensional convolutional neural networks distinguish men and women better than humans do?," Neural Networks (IJCNN)
- [22] <https://news.ycombinator.com/item?id=7928738>
- [23] <https://github.com/clementfarabet/gfx.js/tree/master>
- [24] <https://github.com/torch/torch7/wiki/Cheatsheet>
- [25] SVM : Cortes, C.; Vapnik, V. (1995). "Support-vector networks". Machine Learning
- [26] Xavier Glorot, Antoine Bordes and Yoshua Bengio (2011). "Deep sparse rectifier neural networks".
- [27] <http://www.intechopen.com/books/artificial-neural-networks-architectures-and-applications/applying-artificial-neural-network-hadron-hadron-collisions-at-lhc>
- [28] http://code.cogbits.com/wiki/doku.php?id=tutorial_supervised_3_loss
- [29] http://it.wikipedia.org/wiki/Funzione_di_verosimiglianza
- [30] http://en.wikipedia.org/wiki/Confusion_matrix
- [31] <http://yann.lecun.com/exdb/mnist/>
- [32] http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html
- [33] <http://bleedingedgemachine.blogspot.it/2012/12/gradient-descent.html>
- [34] <http://www.clement.farabet.net/research.html#parsing>
- [35] <https://engineering.purdue.edu/elab/html/teaching-bme495a-computational-neuroscience.html>
- [36] <https://developer.nvidia.com/cuDNN>
- [37] <http://benchmark.ini.rub.de/?section=gtsrb&subsection=news>