Alma Mater Studiorum-Università di Bologna

Scuola di Scienze

Corso di Laurea Magistrale in Informatica

Sviluppo di un sistema basato sul machine learning per l'analisi automatica di immagini ultrasonografiche

Relatore:
Chiar.ma Prof. ELENA LOLI PICCOLOMINI

Correlatore:
Dott. GIOVANNI DI DOMENICO

Presentata da:
MICHAEL CANELLA

Anno accademico 2017-2018

Sessione II

A Martina, alla mia famiglia e a tutti coloro che ci hanno creduto...

Indice

Introduzione

Nel tempo la clinica neurologica ha fatto enormi progressi nella comprensione delle malattie legate al sistema nervoso, trovando soluzioni a diverse patologie e proponendo nuovi metodi di diagnosi e terapie sempre più efficaci. Tuttavia, come in ogni campo, non tutte le sfide sono state ancora vinte, mentre altre si aggiungono, aumentando i punti interrogativi sui quali medici e scienziati concentrano i loro sforzi, al fine ultimo di migliorare e preservare la vita dell'uomo e dare maggiore comprensione del mondo che lo circonda. Una di queste sfide è data da una grave malattia demielinizzante del Sistema Nervoso Centrale (SNC): la *Sclerosi Multipla* (SM). Questa patologia è ben conosciuta per alcuni aspetti, ma ancora oggi non esiste una cura definitiva, anche se con le conoscenze attuali è possibile sottoporsi a terapie in grado di allungare la vita e migliorarne la qualità, in quanto quest'ultima ne viene gravemente afflitta, portando il paziente anche a perdere l'autosufficienza.
Questo lavoro di tesi si concentra su una patologia particolare, l'*Insufficienza Venosa Cronica Cerebrospinale* (CCSVI), scoperta dal medico chirurgo ferrarese Paolo Zamboni e che potrebbe avere una forte correlazione con i pazienti affetti da SM. La patologia ha come peculiarità la formazione di stenosi e malformazioni genetiche o acquisite delle vene cerebrospinali, causando un insufficiente afflusso di sangue al sistema nervoso centrale, aggravando così, con il passare del tempo, le capacità psicomotorie dell'individuo affetto dalla malattia.
Uno degli strumenti più ricorrenti per la diagnosi di questa malattia è l'utilizzo dell'ecografo, poiché permette di osservare in modo non invasivo ed economico i vasi sanguigni, identificando così la presenza di stenosi o flussi irregolari. Il lavoro di tesi propone un software in grado di analizzare i vasi sanguigni nelle immagini ultrasonografiche, in particolare, quelle relative alla vena giugulare interna, un vaso sanguigno principale, più facile da localizzare con l'ecografo e ottimo elemento per osservare particolari disfunzioni nel sistema cardiovascolare.
Il software proposto basa il suo funzionamento sul machine learning, più pre-

cisamente sulle reti neurali, adottando una variante del modello tipo che viene utilizzata principalmente per l'analisi delle immagini, le *reti neurali convoluzionali* (CNN). Tale variante è chiamata *U-net*, il cui nome deriva dalla forma grafica ad "U" che assume il modello e che basa il suo funzionamento "comprimendo" l'immagine in ingresso, estrapolandone alcune caratteristiche e "decomprimendola" in uscita.
Lo scopo ultimo di questo lavoro è di realizzare una rete neurale in grado di riconoscere le sezioni trasversali della vena giugulare interna dalle immagini ultrasonografiche acquisite in B-Mode, affinché se ne possa ricavare un giugulogramma in grado di valutare la dilatazione del vaso tra una pulsazione e l'altra e l'estrapolazione di altri importanti dati e parametri.
In questo elaborato viene presentato dapprima il contesto medico, la patologia in esame nel suo complesso con una piccola digressione sulla vena giugulare interna, sulle modalità di acquisizione delle immagini e alcuni cenni di neurofisiologia. Viene poi presentata la rete neurale convoluzionale, descrivendo alcune delle architetture più popolari e di successo ed entrando nel dettaglio su aspetti come la "regolarizzazione" e la "normalizzazione", nonché la descrizione del modello utilizzato U-net e la sua implementazione in *Tensorflow*, framework di Google appositamente progettato per la costruzione di reti neurali. Infine, viene descritto il processo di addestramento della rete neurale implementata, dal dataset ai miglioramenti effettuati per ottenere il risultato finale, nonché la presentazione dei test realizzati con i parametri usati e i dati ottenuti.

Il lavoro di tesi è articolato nel modo seguente:

Capitolo 1: Descrizione della Sclerosi Multipla e della patologia correlata CCSVI (Insufficienza Venosa Cronica Cerebrospinale), possibile terapia di quest'ultima mediante il metodo Zamboni, descrizione dell'ecografo, delle tipologie di immagini acquisite e nozioni basilari di neurofisiologia;

Capitolo 2: Introduzione alle reti neurali convoluzionali, descrivendone le peculiarità, i principi di funzionamento e ponendo attenzione alle caratteristiche principali per effettuare addestramenti efficaci;

Capitolo 3: Descrizione di alcuni modelli di reti convoluzionali neurali e specifica del modello U-net, descrizione del processo di apprendimento, dall'acquisizione delle immagini per il popolamento dei dataset, ai test di

addestramento, ai passi necessari per l'ottenimento del risultato finale;

Capitolo 4: Presentazione dei risultati ottenuti, commentando i punti di forza e i difetti della rete neurale e dando un riscontro statistico sui dati ottenuti.

Capitolo 1

Cenni di medicina

1.1 Le patologie

L'attenzione di questo lavoro si concentra su due argomenti distinti, Sclerosi Multipla e Insufficienza Venosa Cronica Cerebrospinale, ma con una possibile correlazione ancora in fase di studio.
Nel 2008, il medico chirurgo ferrarese Paolo Zamboni definisce e presenta per la prima volta alla comunità scientifica la definizione della CCSVI [9], descrivendola come una patologia venosa che comporta la comparsa di malformazioni nell'apparato circolatorio, in particolare, nelle vene cervicali e toraciche, causando l'impossibilità di rimuovere in modo efficace il sangue dal sistema nervoso centrale. La CCSVI si presenta con diverse forme di malformazioni, come stenosi, anomalie valvolari o torsioni venose, scatenando effetti collaterali quali ipossia, riduzione del drenaggio dei cataboliti e depositi anomali di ferro lungo le vene cerebrali 1.1. É proprio quest'ultima caratteristica ad accendere il dubbio su una possibile correlazione con la SM, in quanto a questi depositi possono conseguire lesioni e placche, tipiche nella SM. La tipica operazione di diagnosi per la CCSVI prevede l'utilizzo dell'EcoColorDoppler, tecnica non invasiva e sufficientemente efficace per rilevare le anomalie del sistema vascolare. Il trattamento tipico per questa patologia prevede un intervento di angioplastica [1], in cui a partire dalla vena femorale viene inserito un catetere con palloncino e portato fino alla zona d'intervento, in cui il palloncino viene gonfiato, dilatando il vaso sanguigno e ripristinando il funzionamento regolare della vena.
Come già anticipato, la SM o sclerosi a placche è una patologia con alcune caratteristiche comuni con la CCSVI [5]. É la malattia demielinizzante più comune del SNC, che causa quindi una perdita della mielina, una sostanza che funge da

guaina midollare delle fibre nervose. Solitamente questa malattia si manifesta tra i 20 e i 40 anni, ma in casi meno frequenti può essere più precoce o tardiva.

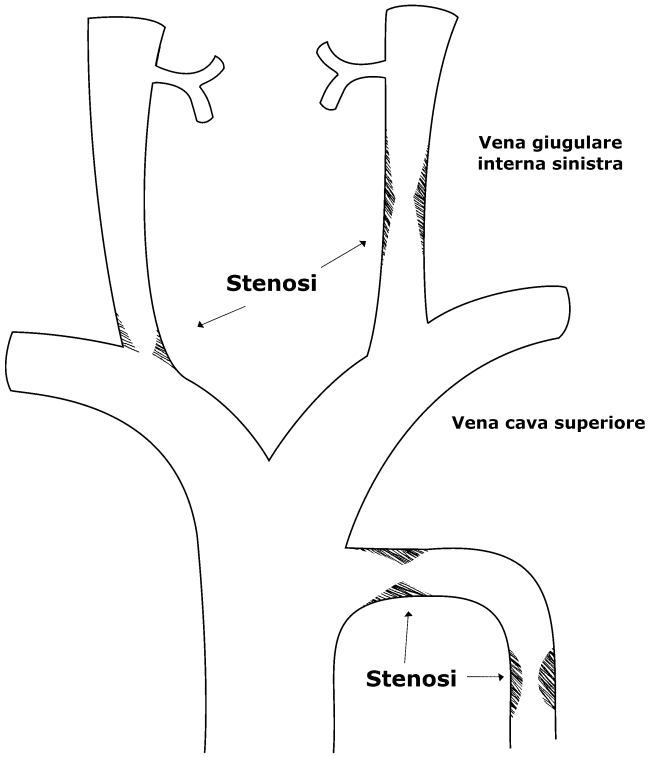


Figura 1.1: Rappresentazione delle stenosi causate dalla CCSVI.

Come anticipato, una delle sue principali caratteristiche è la manifestazione di lesioni e placche, di dimensioni che variano da meno di un millimetro ad alcuni centimetri. Esse colpiscono principalmente i nervi ottici, le zone periventricolari, il tronco encefalico, il cervelletto ed il midollo spinale, causando l'insorgenza di edema ed infiltrazioni venose di linfociti, plasmacellule e macrofagi, da cui ne consegue il calo di produzione di mielina. La sintomatologia connessa alla SM, come si può intuire, è abbastanza variegata, ma dovuta principalmente ai danni neuronali. Le manifestazioni più comuni sono legate ai danni ai nervi ottici, al midollo e al tronco encefalico, con insorgenza di perdita di sensibilità sensoriale e fatica, mentre nei casi più rari si hanno sintomi correlati al lobo frontale del cervello, a causa del quale si ha un deterioramento intellettivo e diversi disturbi mentali. Data la varietà di sintomi e danni la terapia prevede numerose strade, in base agli attacchi manifestati e alla gravità degli stessi, prediligendo l'uso di

corticosteroidi, antidepressivi, immuno-modulatori, immuno-soppressori e fisioterapia.
Nonostante la relazione tra le due patologie, questo lavoro di tesi nasce come supporto per lo studio verso la CCSVI, in quanto si concentra sull'analisi di immagini ecografiche acquisite lungo la vena giugulare interna.

1.2 La vena giugulare interna e il giugulogramma

Il sistema vascolare umano è molto articolato, si occupa di portare sangue ricco di ossigeno ai tessuti periferici e di riportare da questi sangue carico di anidride carbonica al cuore e successivamente ai polmoni, perché venga riossigenato e reimmesso in circolo; alcuni vasi principali sono più importanti di altri.
In questo lavoro di tesi ci si concentra sulla giugulare interna, la vena principale del collo, con il fondamentale ruolo di portare il sangue dal cervello al cuore [2]. Questo vaso sanguigno è la diretta continuazione del seno trasverso e dunque inizia nel forame giugulare alla base del cranio, scende lateralmente alla faringe e posteriormente alla carotide interna, restando sotto al muscolo sternocleidomastoideo; infine termina confluendo nella vena succlavia e formando assieme ad essa il tronco brachiocefalico che si riversa nel cuore. É lunga dai 12 ai 15 cm e presenta due rigonfiamenti (bulbi) alle estremità, uno dei quali, quello inferiore, ha due valvole insufficienti.
Essendo la CCSVI una patologia venosa, lo studio della vena giugulare interna risulta essere un ottimo candidato per una valutazione di salute di un paziente potenzialmente affetto, in quanto una disfunzione in un vaso principale come questo può destare solo preoccupazione. A tal fine sì è preso in considerazione la valutazione del giugulogramma, un grafico che mette in rapporto parametri come l'area del vaso o il flusso sanguigno con il tempo, nei quali le oscillazioni non regolari che si possono riconoscere indicano pazienti non in buona salute.
Il software implementato si occupa del riconoscimento dell'area del vaso nelle immagini ecografiche acquisite in B-Mode, al fine di ricostruire un giugulogramma in modo automatizzato.

1.3 Acquisizione di immagini ultrasonografiche

La CCSVI grava sull'apparato vascolare e per poter diagnosticare efficacemente la patologia viene utilizzato l'Eco-Doppler: è il più indicato per l'analisi e il

monitoraggio di vene e arterie, in grado di fornire immagini sufficientemente chiare, non presenta rischi per il paziente essendo una tecnica non invasiva che utilizza gli ultrasuoni. La più grande limitazione di questo esame è che si tratta di una metodica operatore-dipendente, che richiede grande esperienza e capacità di osservazione da parte del medico chiamato a formulare una diagnosi corretta e coerente. L'intervento richiede il posizionamento di una sonda che emetta ultrasuoni sull'area interessata dopo l'applicazione di un gel tra la sonda e la pelle del paziente, affinché non vi sia aria tra le due parti, permettendo agli ultrasuoni di attraversare i tessuti con maggiore facilità. La sonda emette ultrasuoni a frequenze comprese tra i 2 e i 20 MHz, generate da un cristallo piezoelettrico, ed è possibile regolare l'ampiezza dell'emittente, variando quindi il cono di apertura degli ultrasuoni [3]. La sonda, oltre ad emettere gli ultrasuoni, è anche in grado di riceverli, per lasciarli poi elaborare da un computer che presenta l'immagine su un monitor. L'informazione elaborata è dunque il risultato del tempo impiegato dall'onda emessa a rimbalzare e tornare indietro, successivamente mappata su una scala di grigi che ne indichi la profondità. A causa di una grande differenza d'impedenza acustica tra tessuti e ossa, risulta impossibile attraversare questi ultimi e, anche nel caso in cui ci fossero aria o gas, l'onda sarebbe completamente riflessa, impedendo quindi di vedere oltre quello strato e lasciando una zona d'ombra. Oltre a questa tipica ricostruzione grafica è anche possibile ottenere altre informazioni sfruttando l'effetto Doppler, con il quale si può misurare la velocità del flusso sanguigno integrando le immagini ecografiche a livelli di grigi con informazioni a colori sul flusso (1.2).

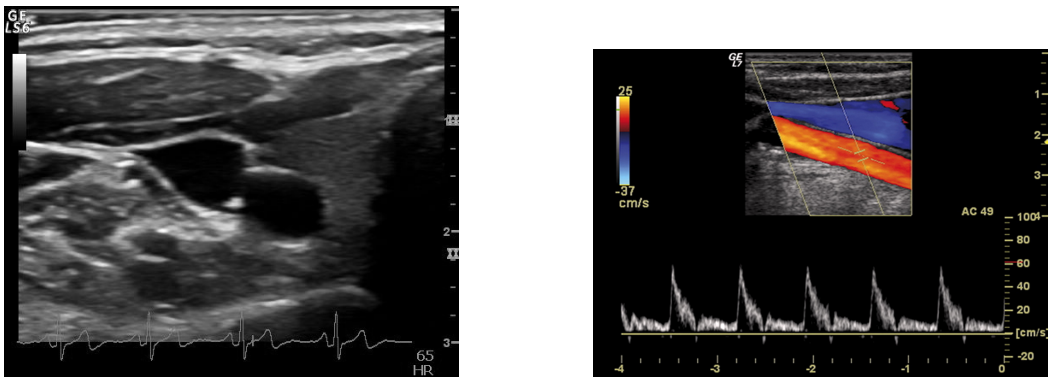


Figura 1.2: A sinistra una tipica immagine ecografica acquisita in B-Mode. A destra un'immagine acquisita tramite EcoColorDoppler.

1.3.1 Sistemi di scansione

Come detto nella sezione precedente, è possibile variare l'ampiezza dell'emittente della sonda, dando la possibilità di ottenere immagini differenti, ma esistono diversi tipi di sonda che permettano la ricostruzione di varie tipologie di immagini. Tutti variano per la disposizione interna dei cristalli e i principali tipi di sonda sono:

- **Lineare**: come si può intuire, la disposizione dei cristalli è regolare lungo la parte piatta della sonda, limitando il campo visivo alla larghezza della sonda. Questa tipologia è idonea a scansioni di superficie, come su collo o arti;
- **Curvilineare**: in questo caso i cristalli sono disposti su una faccia curva, consentendo di avere un campo visivo più ristretto in superficie, ma più ampio in profondità. Questa tipologia presenta dei difetti, ovvero l'immagine risulta di qualità inferiore a causa della minore larghezza del fascio, senza contare i tempi di elaborazione degli echi soggetti a ritardi, in quanto le onde devono percorrere più strada, arrivando più in profondità;
- **Convex**: simile alla tipologia curvilineare, viene prodotta un'immagine a tronco di cono, coprendo un'area più grande;
- **Phased-array**: con i cristalli disposti a ventaglio, questa tipologia viene utilizzata prevalentemente per le scansioni al cuore.

1.3.2 Modalità di acquisizione

In base al tipo di elaborazione effettuata sui dati ottenuti dalle sonde, è possibile ricostruire diversi tipi di immagine, alcuni dei quali senza sfruttare l'effetto Doppler. I possibili tipi di scansione sono:

- **A-Mode**: questa tipologia di scansione opera con la modulazione d'ampiezza (Amplitude-Mode), in cui ogni eco è rappresentato come un picco lungo l'asse verticale, la cui ampiezza rappresenta l'intensità dell'eco stesso;
- **B-Mode**: questo tipo di scansione opera con la modulazione di luminosità (Brightness-Mode), nella quale viene ricostruita un'immagine bidimensionale che in ogni punto rappresenta l'intensità dell'eco rilevato. Ogni linea

verticale mostra una sequenza di echi ricevuti dal trasduttore, dove la posizione di ogni punto rappresenta la distanza dell'eco ricevuto, mentre la luminosità ne mappa l'intensità. L'elaborazione di questo tipo di immagine è piuttosto rapida e con un ritardo trascurabile, consentendo una scansione quasi in tempo reale. Questa tipologia di scansione è quella utilizzata in questo lavoro di tesi;

- **M-Mode**: a partire da un'immagine acquisita in B-Mode è possibile monitorare il movimento dei tessuti (Motion-Mode) tramite questo tipo di scansione. Viene selezionata una linea di scansione dall'immagine in B-Mode e si continua l'acquisizione di quella singola linea, consentendo di ricostruire un'immagine che rappresenta le variazioni degli echi in quella posizione, dando quindi visione dei movimenti dei tessuti. L'immagine ricostruita è dunque bidimensionale, con le linee di scansione sull'asse verticale come nella modalità B-Mode e il tempo sull'asse orizzontale;

- **Spectral Doppler**: con questa tipologia è possibile ricostruire un grafico sfruttando la variazione di frequenza dell'onda riflessa e, quindi, dell'effetto Doppler, ottenendo informazioni circa i flussi sanguigni;

- **Color Flow Imaging**: questo tipo di scansione consente di ottenere informazioni sull'intensità e sulla direzione dei flussi sanguigni, sovrapponendo un'immagine a colori su una acquisita in B-Mode.

Essendo lo scopo di questa tesi fornire uno strumento in grado di poter dare un'analisi sullo stato di salute di una vena giugulare interna, la tipologia d'immagine più conveniente è quella in B-Mode, poiché permette di avere visione della sezione trasversale del vaso e ne permette quindi una valutazione grafica sul grado di dilatazione che presenta tra una pulsazione e l'altra.

1.4 Neurofisiologia

Per poter comprendere al meglio il funzionamento di una rete neurale, questa sezione viene dedicata alle nozioni di base circa la struttura dei neuroni e il loro funzionamento di base, utili a comprendere i meccanismi di input e output. Viene inoltre descritto brevemente il meccanismo di apprendimento del cervello e della rappresentazione degli oggetti che riesce a dare. Tutte queste informazioni sono reperibili per un approfondimento in [4].

1.4.1 I neuroni

Il neurone è una cellula che contribuisce in modo fondamentale alla formazione del sistema nervoso e possiede le importanti caratteristiche chimiche e fisiologiche di ricevere, integrare e trasmettere impulsi nervosi. Esso è costituito essenzialmente da tre parti: un corpo cellulare, all'interno del quale risiede il nucleo e gli altri organi che garantiscono le principali funzioni cellulari; i dendriti, estensioni del corpo che ricordano vagamente le diramazioni di un albero e che hanno l'importante funzione di ricevere segnali; l'assone, altro prolungamento del corpo centrale e incaricato della trasmissione del segnale verso le altre cellul (1.3)e.

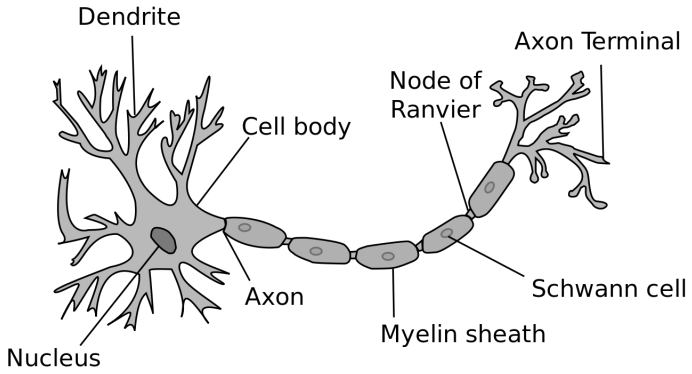


Figura 1.3: Rappresentazione grafica di un neurone.

La giunzione asso-dendritica viene chiamata sinapsi ed è il punto di passaggio dell'informazione da un neurone all'altro e ogni neurone può formarne circa 1000 e riceverne anche 10000. A onor del vero, esistono altri tipi di sinapsi, come quelle asso-assoniche e asso-somatiche, tuttavia la maggior parte delle giunzioni sono del primo tipo, ovverosia asso-dendritiche. Quando avviene il passaggio d'informazione in una sinapsi parliamo di trasmissione sinaptica e si può distinguere in due tipi: chimica ed elettrica. La principale differenza tra le due è che nella prima il collegamento tra i due neuroni non è completo, ovvero c'è un piccolo spazio chiamato fessura sinaptica e il passaggio della corrente causa il rilascio di una sostanza trasmettitrice, mentre nell'altra tipologia il contatto tra i due neuroni è diretto tramite particolari canali ionici, detti giunzioni comunicanti, in cui il passaggio della corrente è diretto. I segnali ricevuti dal neurone sono quindi molteplici ed esso provvede ad integrarli tutti; il risultato prodotto viene inviato come risposta se tale eccitazione supera la soglia minima di attivazione di quel neurone.

1.4.2 Il riconoscimento degli oggetti nel cervello

Ancora oggi non è completamente chiaro come il cervello sia in grado di riconoscere gli oggetti visivamente, tuttavia esistono svariate considerazioni in merito che permettono di capire come il cervello interpreti le percezioni ricevute. Per convenzione e comodità il cervello viene suddiviso in regioni, chiamate aree di Brodmann, ognuna delle quali contiene gruppi cellulari dedicati all'elaborazione di una specifica informazione. La regione deputata agli stimoli visivi è l'area 17 di Brodmann, chiamata anche corteccia visiva, corteccia striata o V1 e comprende anche sottosezioni più piccole chiamate aree visive corticali extra-striate V2, V3, V4 e V5, le quali però si estendono anche in altre aree di Brodmann.
La progressione dell'informazione visiva è una materia assai affascinante ed estremamente complessa che per dovere di sintesi può essere semplificata in questo modo: lo stimolo retinico scorre attraverso due vie parallele fino alla corteccia visiva, da questa poi continuano attraverso due vie corticali extrastriate separate, una dorsale e una ventrale; la prima reagisce particolarmente al movimento, mentre la seconda ai contrasti e alle sagome.
La via corticale ventrale si estende da V1 a V2 a V4 raggiungendo la corteccia inferotemporale. Le cellule di V2, insieme a quelle di V1, sono sensibili all'orientamento, al colore e alla disparità retinica degli stimoli visivi e proseguono l'analisi delle sagome degli oggetti che già viene iniziata in V1. Le cellule di V4 esibiscono sensibilità verso i colori, tuttavia alcune zone di V4 sono sensibili anche a combinazioni di forme e colori e contribuiscono ad eccitare la corteccia inferotemporale, area in cui sembra avere luogo la maggior parte dell'elaborazione per quanto riguarda il riconoscimento delle facce e altre forme complesse.
Tutto questo elenco geografico di stimoli serve a capire come i neuroni del cervello rispondano alla rilevazione di forme basilari, riconoscendo piccoli pattern, che una volta integrati insieme permettono di identificare gli oggetti; questo assomiglia, nel campo delle reti convoluzionali neuronali, all'utilizzo di una feature map, in cui la rete neurale apprende alcune caratteristiche specifiche delle immagini che elabora, al fine di riconoscere e classificare gli oggetti al suo interno.

1.5 Reti neurali artificiali

Le reti neurali artificiali rappresentano una nicchia nel campo dell'apprendimento automatico, che negli anni sta prendendo sempre più piede tra gli scienziati e gli sviluppatori, in quanto stanno dimostrando di essere capaci di grandi poten-

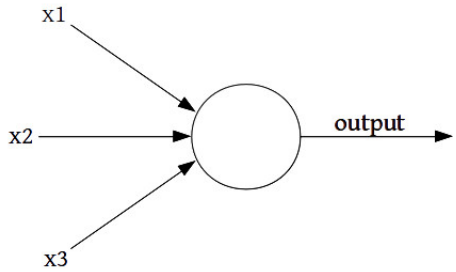


Figura 1.4: Rappresentazione di un percettrone, tipologia di neurone artificiale.

zialità per la risoluzione di numerosi problemi in molte aree d'interesse: automobilistica, meteorologia, marketing, medicina e tante altre. Esse sono l'espressione di un modello matematico ispirato ad una reale rete neurale biologica, come il cervello umano, fornendo una rappresentazione dei neuroni interconnessi tra loro secondo opportuni modelli e supportato da un sistema di "pesi" che, opportunatamente impostati, consentono di veicolare le informazioni verso un risultato piuttosto che un altro, portando a compimento il processo di apprendimento.
Per comprenderne i principi di funzionamento si può fare riferimento al *percettrone* (o perceptron) [11], un tipo di neurone artificiale ideato tra il 1950 e 1960 da Frank Rosenblatt. Il percettrone funziona ricevendo alcuni input binari, producendo un unico output binario calcolato sulla base di alcuni pesi, numeri reali che esprimono l'importanza dei rispettivi input e output (Figura 1.4).
Il neurone presenta in uscita un valore, 0 o 1, sulla base della posizione del risultato della somma pesata tra i valori di input e i pesi, rispetto ad una soglia. Ponendo x come valore di input e j come valore di un peso l'output del neurone è:

$$output = \begin{cases} 0 & se \sum_j w_j x_j \leq soglia \\ 1 & se \sum_j w_j x_j > soglia \end{cases} \tag{1.1}$$

In un modello di questa semplicità è facile dedurre come il valore di un peso possa determinare l'importanza del proprio input correlato e condurre il risultato verso un esito piuttosto che un altro. L'unione di più elementi di questo tipo formano una rete neurale in grado di essere più espressiva, in cui gli output dei primi neuroni diventeranno gli input di quelli successivi e così via fino al risultato finale (Figura 1.5).

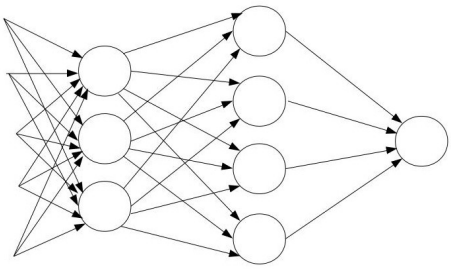


Figura 1.5: Rappresentazione di una rete di percettroni a tre livelli.

La notazione utilizzata per descrivere il percettrone nella Formula 1.1 può essere semplificata ulteriormente in modo tale da facilitarne la lettura, ma soprattutto aprire la possibilità di poter descrivere altri tipi di neuroni che verranno introdotti nel successivo capitolo. Per la semplificazione è sufficiente sostituire la sommatoria con un prodotto scalare e spostare la soglia dall'altra parte dell'uguaglianza introducendo quello che viene chiamato *bias*:

$$output = \begin{cases} 0 & se\ w \cdot x + b \leq 0 \\ 1 & se\ w \cdot x + b > 0 \end{cases} \tag{1.2}$$

Si può intuire come il termine bias possa essere determinante nel "decidere" se un neurone debba attivarsi o meno.
Un tipico esempio didattico per comprendere il funzionamento di una rete neurale è quello del riconoscimento dei numeri scritti a mano, riconducibile ad un classico problema di classificazione. La rete per poter appredere necessita di un insieme di dati per l'apprendimento, denominato *data set*, dalla quale attingerà durante l'addestramento. Per questo esempio ci si può ricondurre al MNIST data set [12], popolato da decine di migliaia di immagini scansionate di numeri scritti a mano, ognuna di esse con la propria classificazione corretta (Figura 1.6).

Figura 1.6: Esempio di immagini nel data set MNIST con numeri scritti a mano scansionati.

Le immagini in questione sono in scala di grigi e di dimensione 28x28 pixel e sono separate in due insieme, uno da 60.000 immagini come dati da addestramento e l'altro da 10.000 immagini usate per i test e valutare quanto la rete è brava a riconoscere i numeri.

Si può porre come input $x = 28 \times 28$, un vettore di 784 valori contenenti i valori di ogni intensità dei pixel dell'immagine e si può denotare l'output come $y = y(x)$ dove y è un vettore di dimensione 10 (uno per ogni numero riconoscibile, quindi da zero a nove). A partire da queste informazioni è possibile modellare l'architettura della propria rete neurale, decidere di quanti strati comporla e di che dimensione; sfruttando qualche euristica del caso è possibile ribaltare l'esito delle performance della rete.
Per poter definire davvero un apprendimento occorre sapere quanto si stia sbagliando ed è quindi necessario introdurre una **funzione di costo** (anche chiamata di perdita o obiettivo) per poter variare nella direzione giusta i pesi e i bias:

$$C(w,b) \equiv \frac{1}{2n} \sum_{x} \|y(x) - a\|^2 \tag{1.3}$$

dove w e b rappresentano l'insieme di pesi e bias, mentre n è il numero totale degli input e a è il vettore degli output della rete.
C rappresenta l'*errore quadratico medio*, o funzione del costo quadratico e risulta non essere mai negativo dato che i termini della somma non sono negativi e dunque, un valore vicino allo zero indica che $y(x)$ è molto simile all'output a e che l'attuale combinazione di pesi e bias è conveniente. Diventa quindi di fondamentale importanza minimizzare il valore ottenuto da questa funzione e in ausilio a tale compito è possibile utilizzare il gradiente, il cui calcolo permette di ottenere un vettore che indica la "direzione" da prendere per il bilanciamento dei pesi e del bias, sfruttando le derivate parziali che permettono di valutare lo spostamento dei valori in una direzione e correggerla nel caso in cui ci si stia allontanando dal minimo della funzione di costo. Tale operazione viene chiamata **discesa del gradiente** e per farla funzionare a dovere occorre scegliere un *valore di apprendimento* adatto. Tale valore indica di variare i valori in direzione del gradiente ed è immediato comprendere come un valore troppo grande si rischierebbe di non centrare il minimo e di sorpassarlo, dando luogo ad una serie di fluttuazioni attorno a quest'ultimo. Un valore troppo piccolo invece potrebbe non far convergere mai la rete. Una rappresentazione grafica della discesa del gradiente è data dalla Figura 1.7 in cui C è funzione di due variabili $v1$ e $v2$ e il gradiente è rappresentato dalla freccia verde; lo scopo è quello di far "scivolare" la palla verso il fondo, o minimo globale. Le informazioni di questo paragrafo possono essere approfondite in [13] e [14]

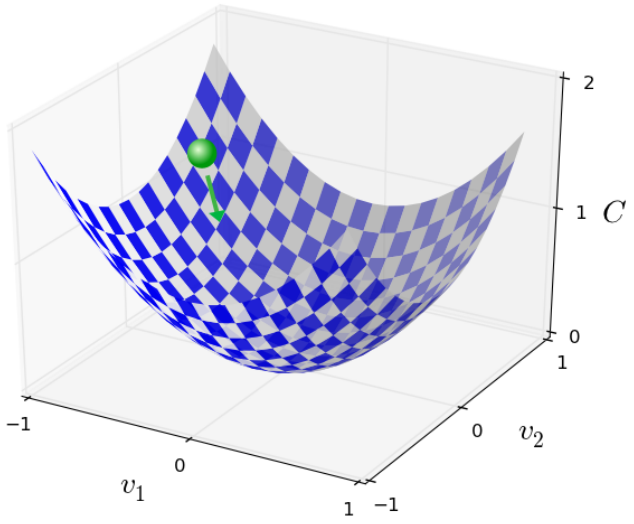
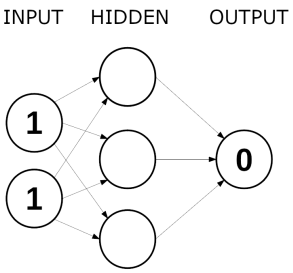


Figura 1.7: Discesa del gradiente.

1.6 Una semplice rete neurale artificiale

Per offrire maggiore compresione sul funzionamento di questo oggetto matematico si faccia riferimento al seguente esempio pratico in cui si addestra una semplice rete neurale per funzionare come l'operatore logico XOR il quale ritorna il valore 1 se i due dati in ingresso sono diversi (Figura 1.6).



Si può ricondurre il meccanismo della rete neurale a due processi principali: la **propagazione in avanti** e la **propagazione all'indietro** e l'addestramento non è altro che la ripetizione di questi due passi al fine di bilanciare i pesi fino all'ottenimento del risultato desiderato. Per propagazione in avanti si intende l'appli-

cazione dei pesi ai dati in ingresso dei neuroni per elaborare gli output, mentre con la propagazione all'indietro si cerca di inferire dall'errore commesso dei nuovi pesi da offrire alla rete. Riprendendo l'esempio dell'operatore XOR si ipotizzi di assegnare alle connessioni dei dati di input dei pesi ottenuti in modo casuale tramite una distribuzione normale gaussiana (Figura 1.8)

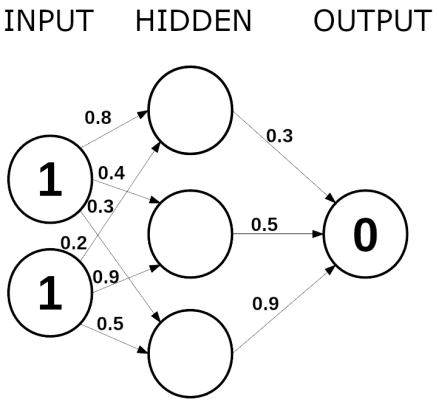


Figura 1.8: Operatore XOR su una rete neurale. Alle connessioni dei dati in ingresso sono assegnati dei pesi.

Si calcolino a questo punto i primi valori dello strato nascosto applicando la formula 1.1, omettendo per un attimo la condizione sulla soglia, ottenendo quindi i valori temporanei:

1 * 0.8 + 1 * 0.2 = 1
1 * 0.4 + 1 * 0.9 = 1.3
1 * 0.3 + 1 * 0.5 = 0.8

Con questi risultati si potrebbe applicare ora il criterio di soglia dell'equazione 1.1 per definire i valori dei neuroni nascosti, tuttavia per usare un criterio più "evoluto" si può utilizzare la funzione di attivazione *Sigmoid*, descritta in modo più approfondito nel prossimo capitolo e che ci consente ora di calcolare i valori dei neuroni nascosti. La funzione Sigmoid $\sigma(x)$ è definita come $\sigma\left(x\right) = 1/\left(1 + e^{-x}\right)$ ed applicata ai valori calcolati precedentemente permette di ottenere (Figura 1.6):

$\sigma(1.0) = 0.7310$
$\sigma(1.3) = 0.7858$
$\sigma(0.8) = 0.6899$

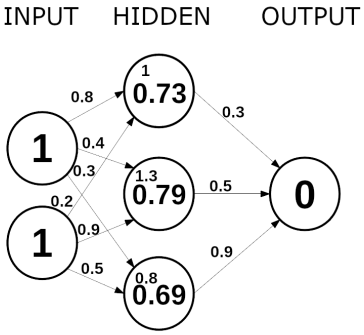


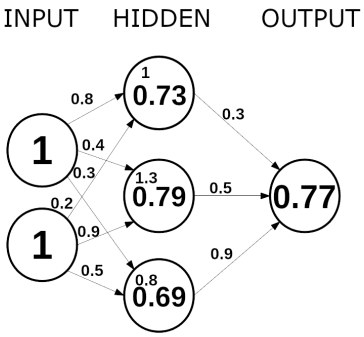
Figura 1.9: I valori dei neuroni nascosti sono stati calcolati tramite la funzione Sigmoid.

La somma dei prodotti dei valori dello strato nascosto è:

$sum = 0.73 * 0.3 + 0.79 * 0.5 + 0.69 * 0.9 = 1.235$

ed applicata la funzione Sigmoid si ottiene il risultato:

$\sigma(1.235) = 0.7746$



Tale risultato non è soddisfacente poichè quel valore finale è più vicino ad un 1 che ad uno 0, ed occorre quindi apportare delle modifiche ai pesi della rete neurale e per fare ciò si procede con la valutazione dell'errore commesso ed iniziando

quindi la fase di propagazione all'indietro. L'errore commesso può essere definito facilmente come $err = obiettivo - calcolato$, cioè $err = 0 - 0.77 = -0.77$ e riprendendo le somma dei valori dello strato nascosto e il risultato di output si può utilizzare la derivata prima della funzione Sigmoid per stabilire la variazione da apportare ai pesi (si possono ovviamente utilizzare criteri differenti):

$$S'(sum)\frac{dsum}{dresult} \tag{1.4}$$

Con sum la somma dei valori dello strato nascosto e $result$ il valore di output. La variazione dei pesi è così calcolabile applicando a questa derivata il margine di errore:

$$S'(sum)\frac{dsum}{dresult} \times (obiettivo - calcolato) = \Delta sum \tag{1.5}$$

e quindi:

delta_sum = σ'(1.235) * (-0.77)
delta_sum = -0.1343

Volendo applicare questa variazione al resto della rete si procede all'indietro ricavando dapprima i nuovi pesi dei neuroni nascosti, tenendo a mente queste tre relazioni:

$$H_{result} \times w_{h->o} = O_{sum}$$

$$\frac{dO_{sum}}{dw_{h->o}} = H_{result} \tag{1.6}$$

$$dw_{h->0} = \frac{dO_{sum}}{H_{result}}$$

In cui H_{result} sono i valori calcolati sugli strati nascosti, $w_{h->o}$ sono i valori dei pesi tra lo strato nascosto e quello di output e O_{sum} è il valore calcolato sul neurone di output prima di applicare la funzione Sigmoid. Quindi:

```
# variazione da applicare ai pesi dei neuroni nascosti
delta_weights = delta_output_sum / hidden_layer_results
delta_weights = −0.1344 / [0.7310, 0.7858, 0.6999]
delta_weights = [−0.1838, −0.1710, −0.1920]

# nuovi pesi dei neuroni nascosti
new_hw1 = 0.1162
new_hw2 = 0.329
new_hw3 = 0.708
```

Da notare come i pesi che danno contributo maggiore hanno anche una variazione maggiore. Sfruttando le relazioni di Formula 1.6 e con l'utilizzo di un po' di algebra si possono ottenere i nuovi valori per i neuroni nascosti:

$$dH_{sum} = \frac{dO_{sum}}{w_{h->o}} \times S'\left(H_{sum}\right) \tag{1.7}$$

dai si può calcolare:

```
# variazione neuroni nascosti = variazione output / pesi neuroni nascosti
# ∗ derivata sigmoide dei valori delle somme dei neuroni nascosti
delta_h = delta_output_sum / hw ∗ S'(h_sum)
    delta_h = −0.1344 / [0.3, 0.5, 0.9] * S'([1, 1.3, 0.8])
delta_h = [−0.448, −0.2688, −0.1493] ∗ [0.1966, 0.1683, 0.2139]
delta_h = [−0.088, −0.0452, −0.0319]
```

Tali valori servono per ribilanciare i pesi dei neuroni d'ingresso, sfruttando l'equazione:

$$I \times w_{i->h} = H_{sum} \tag{1.8}$$

da cui:

$$\frac{dH_{sum}}{dw_{i->h}} = I \qquad => \qquad dw_{i->h} = \frac{dH_{sum}}{I} \tag{1.9}$$

calcolando si ottiene:

```
input1 = 1
input2 = 1

delta_w = delta_hidden_sum / input_data
delta_w = [−0.088, −0.0452, −0.0319] / [1, 1]
delta_w = [−0.088, −0.0452, −0.0319, −0.088, −0.0452, −0.0319]

new_w1 = 0.712
new_w2 = 0.3548
new_w3 = 0.2681
new_w4 = 0.112
new_w5 = 0.8548
new_w6 = 0.4681
```

A questo punto è possibile propagare i pesi nuovamente in avanti con i passaggi descritti all'inizio di questo paragrafo per arrivare alla fine di quest'altra iterazione ed ottenere al neurone di output il valore 0.69 e scoprire di essere ancora lontani dalla soluzione, ma sicuramente più vicini. Le reti neurali richiedono numerose iterazioni per raggiungere la soluzione e questo problema, seppur semplice, non è da meno. Nel prossimo capitolo vengono descritti i particolari più salienti delle reti neurali convoluzionali approfondendo le caratteristiche citate in questi ultimi due paragrafi.

Capitolo 2

Reti Neurali Convoluzionali

2.1 Introduzione

In questo lavoro di tesi si utilizza una particolare tipologia di rete neurale, la rete neurale convoluzionale (Convolutional Neural Network o CNN), specializzata nella risoluzione di un precisa categoria di problemi: la classificazione degli oggetti in un'immagine, ovvero la determinazione e la distinzione degli oggetti presenti nella scena, che siano persone, animali o cose. Questi problemi posso diversificarsi ulteriormente in base all'obiettivo da raggiungere, per esempio, nel caso di questo lavoro si utilizza l'approccio di segmentazione dell'immagine per distinguere un vaso sanguigno dal resto dell'immagine, mappando i pixel in una mappa di probabilità, indicando quanto un punto potrebbe rappresentare un vaso sanguigno o meno. Inoltre, questa tipologia di rete neurale viene usata anche per altri tipi di problemi, non solo legati al riconoscimento nelle immagini video, ma anche nel linguaggio naturale, nella bioinformatica e altri.

2.2 L'operazione di convoluzione

Le CNN prendono il nome dall'operatore di convoluzione [15] di cui si avvalgono. In generale, quest'operatore mette in relazione due funzioni con numeri reali per ottenerne una terza che indichi quanto una sia modificata dall'altra, ed esiste per ogni funzione per cui il seguente integrale è definito:

$$s(t) = \int_{-\infty}^{\infty} x(a)\, w(t-a)\, da. \tag{2.1}$$

Si tratta dunque dell'integrale del prodotto tra due funzioni, di cui una è invertita e spostata ed è denotato più semplicemente con un asterisco:

$$s(t) = (x * w)(t). \tag{2.2}$$

Nella terminologia delle reti convoluzionali neurali, il primo argomento viene indicato come **input**, il secondo come **kernel** e l'output come **feature map**.
Considerando il caso d'uso di questo lavoro, i valori manipolati non sono numeri reali, bensì interi, poiché le intensità dei punti che costituiscono un'immagine sono solitamente rappresentate con questi tipi di numeri e quindi è possibile definire la convoluzione discreta come:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)\, w(t-a). \tag{2.3}$$

Inoltre, si possono utilizzare le convoluzioni su più assi e ciò è conveniente per operare, per esempio, su immagini bidimensionali, stabilendo $\boldsymbol{I}$come immagine di input e $\boldsymbol{K}$come kernel bidimensionale si può riscrivere la convoluzione come:

$$S(i,j) = (I * K)(i,j) = \sum_{m}\sum_{n} I(m,n)\, K(i-m, j-n). \tag{2.4}$$

Tuttavia, molte CNN implementano una funzione simile alla convoluzione chiamata **cross-correlation**, la quale evita di invertire il kernel:

$$S(i,j) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i+m, j+n)\, K(m,n). \tag{2.5}$$

L'adozione di questo operatore è dovuta ad alcune sue vantaggiose caratteristiche. Le reti neurali tradizionali comportano un'interazione completa tra input e output, cioè ad ogni ingresso corrisponde un'uscita e questa limitazione può essere aggirata tramite l'utilizzo della convoluzione, utilizzando un kernel più piccolo rispetto all'input. In questo modo è possibile analizzare milioni di pixel rilevando alcune piccole caratteristiche significative, come i bordi, utilizzando un kernel di dimensioni molto più limitate rispetto all'immagine, riducendo quindi l'utilizzo di memoria e di operazioni necessarie a produrre l'output.
Un'altra possibilità che offre la convoluzione è la **condivisione dei parametri** in cui un peso non è legato ad un singolo livello della rete, ma può essere riutilizzato altrove, evitando quindi di calcolare kernel completamente nuovi e aumentare l'efficienza totale rispetto alle reti tradizionali completamente connesse.
Infine, la condivisione dei parametri permette alla convoluzione di acquisire la proprietà di **equivarianza** alla traslazione; se l'input di una funzione cambia e l'output varia nello stesso modo, allora quella funzione è equivariante:

$$f\left(g\left(x\right)\right) = g\left(f\left(x\right)\right) \tag{2.6}$$

2.3 Architettura generale della CNN

La CNN è composta essenzialmente da due strati di neuroni, di input e di output, separati da altri strati nascosti, i quali si occupano principalmente di operazioni di convoluzione, pooling e normalizzazione [16]. Le parti fondamentali di cui sono composte le CNN si possono quindi riassumere in:

- **Strati convoluzionali**: Si occupano di applicare un'operazione convoluzionale all'input ricevuto e trasferiscono il risultato agli strati successivi, emulando la risposta di un neurone allo stimolo visivo. L'input ricevuto è solitamente limitato al campo di ricezione del singolo neurone, permettendo di frammentare l'immagine in più aree e distribuire il lavoro su più neuroni e consentendo di poter riconoscere particolari pattern, utili per la classificazione delle immagini;

- **Strati di pooling**: Questi strati permettono di fare una valutazione sugli output dei neuroni. Un tipico esempio è dato dal max pooling, in cui viene scelto il valore massimo di output del neurone. Tale operazione risulta essere **invariante** alle piccole traslazioni dell'input, ovvero se l'input subisce una piccola traslazione, il risultato di max pooling rimane uguale;

- **Strati completamente connessi**: E' una particolare configurazione di interconnessione tra neuroni in cui tutte le unità sono connesse con le altre;

- **Pesi**: Solitamente sono espressi come un vettore di valori in congiunzione con un bias, i quali vengono impiegati nelle funzioni utilizzate dai neuroni per processare l'input. Tramite la variazione di questi valori in modo opportuno la rete è in grado di avanzare correttamente nel processo di apprendimento.

Oltre a queste proprietà, le CNN possiedono anche alcune caratteristiche comuni alle altre reti neurali non convoluzionali.

2.4 Funzione di attivazione

E' una funzione che compie alcune operazioni matematiche e stabilisce se il neurone debba attivarsi o meno, tipicamente nell'intervallo [0,1] o [-1,1], un valore più vicino a 1 fa sì che il neurone si attivi. Le funzioni più comuni utilizzate sono:

- **Sigmoid**: Segue la formula matematica $\sigma(x) = 1/(1 + e^{-x})$ e dato un numero reale, ne ritorna uno compreso tra 0 e 1. La particolarità della funzione sta nel "convertire" valori molto negativi a 0 e quelli molto positivi a 1, avvicinandosi molto all'interpretazione del tasso di attivazione di un vero neurone. Tuttavia la sua adozione non è più comune in quanto manifesta due effetti collaterali:

  1. **Saturazione e "spegnimento" del gradiente**: accade quando i valori di attivazione si polarizzano sulle estremità 0 e 1, valori per i quali il gradiente rimane quasi a zero, lasciando spazio ad un apprendimento minimo;
  2. **I valori di ouptut non sono centrati rispetto allo zero**; simile al problema precedente, questo si verifica durante il processo di backpropagation, in cui se i dati di input di un neurone rimangono positivi, il gradiente applicato sui pesi rimarrà sempre positivo o negativo, con conseguente fluttuazione dei valori di aggiornamento dei gradienti e dei pesi, rischiando quindi di convergere lentamente alla soluzione, o non convergere affatto.

- **Tanh**: La sua funzione matematica è $tanh(x) = 2\sigma(2x) - 1$ e "spalma" il valore in ingresso tra -1 e 1 e presenta lo stesso problema di saturazione della funzione Sigmoid; tuttavia questo effetto collaterale viene mitigato dalla possibilità di avere valori centrati rispetto allo zero ed è quindi da preferire a Sigmoid.

- **ReLU**: Acronimo di Rectified Linear Unit, è una delle funzioni più popolari. Risponde alla funzione $f(x) = max(0, x)$, per cui i valori $x < 0$ vengono tagliati a 0. Questa funzione risulta essere efficace nell'accelerare considerevolmente la discesa stocastica del gradiente senza saturare quest'ultimo, oltre a sfruttare una funzione meno costosa dal punto di vista computazionale. Tuttavia, durante la fase di addestramento le unità che implementano ReLU possono "morire" a causa di gradienti eccessivamente grandi,

inducendo il neurone a rimanere continuamente spento. Questo problema può essere aggirato utilizzando un parametro di learning rate non troppo grande.

- **Leaky ReLU**: Questa funzione tenta di risolvere il problema dello spegnimento di ReLU, consentendo una piccola pendenza negativa (e.g. 0.01). Utilizza la funzione $f(x) = \alpha\ min(0, x) + max(0, x)$. I risultati di questa funzione non sono consistenti e sono probabilmente specifici del problema affrontato.

Un confronto visivo tra le differenti funzioni può essere visto in Figura 2.1

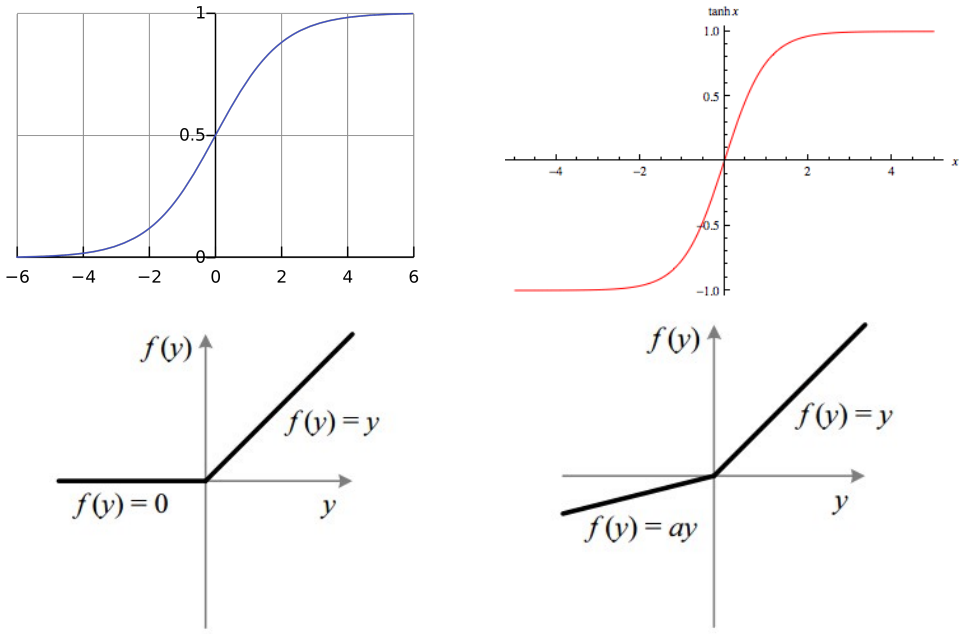


Figura 2.1: In alto a sinistra Sigmoid; in alto a destra Tanh; in basso a sinistra ReLU; in basso a destra Leaky ReLU.

2.5 Gradiente

Le reti neurali si avvalgono generalmente di una funzione di perdita, o loss function, per valutare la qualità delle proprie prestazioni. Tale informazione viene quindi usata per bilanciare nel modo corretto i pesi utilizzati per l'addestramento. Esistono diverse possibilità su come regolare i pesi, la più banale e meno efficace è senza dubbio l'utilizzo di pesi casuali, che porta a lunghe e inconcludenti

ricerche. Un affinamento di tale tecnica è dato da quella che si chiama ricerca locale casuale, nella quale a partire da pesi casuali si generano delle perturbazioni casuali di pesi e se tale perturbazione abbassa il valore di perdita allora i pesi verranno aggiornati con quelli. Questo approccio migliora la situazione, ma di poco, oltre ad essere particolarmente costoso in termini computazionali. Una terza soluzione sicuramente più efficace è data dal calcolo del gradiente, calcolabile in due modi [10]:

- **Gradiente numerico**: Questo metodo è sicuramente più semplice, ancorché più lento. Essenzialmente si tratta di calcolare il gradiente iterativamente che per definizione è un vettore le cui componenti sono le derivate parziali della funzione. La "direzione" del gradiente da seguire è solitamente quella negativa, per abbassare la funzione di perdita. Questa soluzione può richiedere tempi troppo lunghi, in quanto si considera solo la direzione da prendere, ma non l'entità dello spostamento; con un passo troppo piccolo si rischia di non convergere o di impiegare molto tempo per farlo, mentre con un passo troppo lungo si corre il pericolo di allontanarsi dal punto di convergenza talmente spesso da allungare il tempo per trovare la soluzione;
- **Gradiente analitico**: Questa soluzione permette di derivare una formula diretta per il gradiente, senza approssimazione e veloce da calcolare, tramite l'analisi matematica. Di contro, questo metodo è più soggetto ad errori e si tende quindi a confrontarlo con il gradiente numerico per assicurarne la correttezza. Per dare un esempio pratico, si consideri la funzione di perdita per le macchine a vettori di supporto (o SVM, utilizzate per i problemi di classificazione):

$$L_i = \sum_{j \neq y_i} \left[max\left(0, w_j^T x_i - w_{y_i}^T x_i + \Delta\right) \right] \tag{2.7}$$

Si può differenziare la funzione rispetto ai pesi, per esempio w_{y_i}:

$$\nabla_{w_{y_i}} L_i = -\left(\sum_{j \neq y_i} 1\left(w_j^T x_i - w_{y_i}^T x_i + \Delta > 0\right) \right) x_i \tag{2.8}$$

Ciò che si ottiene è un'espressione che una volta implementata conti il numero di classi che non abbiano soddisfatto la condizione richiesta e che quindi contribuiscano all'incremento della funzione di perdita e dunque il vettore dei dati x_i scalato con il numero calcolato è il nuovo gradiente.

2.5.1 Discesa del gradiente

La procedura che si occupa di valutare ripetitivamente il gradiente per poi aggiornare i parametri in modo coerente si chiama *discesa del gradiente*. L'idea di base è semplice e può essere definita con poche righe di codice Python:

```
while True:
    weights_gradient = evaluate_gradient(loss_f, data, w)
    weights += - step_size * weights_gradient
```

Ovviamente le implementazioni possono variare, ma il concetto di base rimane questo. E' importante scegliere con cura il valore di **learning rate** che indica di quanto "spostarsi" in direzione del gradiente.
Ci sono casi in cui i dati di addestramento sono di grandi dimensioni, dell'ordine dei milioni e la computazione della funzione di perdita su tutto il set risulta troppo costosa per l'aggiornamento di un singolo parametro. In questi casi il gradiente viene calcolato utilizzando modeste quantità di dati rispetto all'intero set; questi gruppi vengono chiamati **batches** e la loro efficacia si dimostra quando le immagini utilizzate siano praticamente identiche oppure quando siano presenti alcune lievi operazioni di trasformazione. In un caso di questo tipo il calcolo del gradiente risulta identico per tutte le immagini simili e la perdita calcolata rimane la stessa che si ottiene facendo i calcoli su un sottoinsieme di quelle immagini. Dunque si può affermare che il calcolo del gradiente di un "mini-batch" è una buona approssimazione del gradiente dell'intero obiettivo. Questa tecnica viene definita **Mini-batch gradient descent**.
Si può considerare il particolare caso in cui il mini-batch contenga un singolo elemento e in cui la computazione del gradiente risulti in genere meno efficiente, in quanto risulta più performante calcolarne uno su più esempi, piuttosto che più volte sullo stesso esempio; tale metodo si chiama **Discesa del Gradiente Stocastico**.

2.6 Impostazioni dei dati e della rete neurale

Molte volte per ottenere dei risultati non basta avere dei dati e avviare la rete neurale, ma è necessario effettuare delle operazioni sui dati e scegliere dei parametri con cura, che consentano alla rete di convergere il più velocemente possibile. Vengono illustrate ora alcune delle più comuni operazioni.

2.6.1 Elaborazione dei dati

Un buon punto di partenza può essere quello di svolgere alcune operazioni sui dati prima di iniziare l'elaborazione. Un primo passaggio può essere la **sottrazione della media**, tecnica che si adopera sottraendo il valor medio dell'immagine a tutti i valori dei pixel dell'immagine per ogni pixel considerato, eventualmente sui tre canali dei colori.
Un'altra operazione comune è la **normalizzazione**, un'operazione che consente di mettere sulla stessa scala le dimensioni dei dati. Si può ottenere semplicemente dividendo ogni dimensione alla quale abbiamo sottratto il valor medio per la sua deviazione standard.

2.6.2 Inizializzazione dei pesi

Prima di avviare l'addestramento di una rete neurale è opportuno inizializzare i parametri relativi ai pesi e, per fare ciò, si possono adottare diverse soluzioni. Si può iniziare ad affrontare questo problema considerando la soluzione più semplice (e totalmente sconsigliata): inizializzazione dei parametri a zero. Questa mancanza di asimmetria tra i neuroni può portare questi a calcolare lo stesso output e quindi a computare anche lo stesso gradiente, cosa che può impedire di trovare la convergenza della rete. Questa considerazione suggerisce quindi di non utilizzare gli stessi valori di inizializzazione per i pesi.
Un altro tentativo può essere quello di inizializzare i pesi con valori piccoli, vicini allo zero, in modo tale da rompere la simmetria e quindi permettere ai neuroni di computare la propria via da seguire. Questo metodo è migliore, ma potrebbe portare ad una lentissima convergenza, in quanto il gradiente è calcolato considerando i pesi, i quali essendo molto piccoli rendono l'avanzamento del gradiente minimo. Poco proficua sarebbe lo stesso tipo di inizializzazione, ma con numeri grandi, portando quindi all'esplosione del gradiente.
Queste soluzioni proposte soffrono del problema per cui la distribuzione degli

output, a partire da un'inizializzazione casuale, ha una varianza che cresce con la dimensione del numero di input. Si può quindi ovviare il problema normalizzando la varianza di ogni output a uno, scalando i suoi pesi della radice quadrata del numero dei suoi input. Una semplice implementazione Python usando Numpy può essere:

```
w = np.random.randn(n) / sqrt(n)
```

dove n è il numero di input. In questo modo tutti i neuroni hanno la stessa distribuzione di output aumentando le possibilità di convergenza. Esiste un caso specifico di questa implementazione, nonché anche più popolare, dedicato ai neuroni di tipo ReLU, in cui la varianza dei neuroni dovrebbe essere $2/n$ e quindi:

```
w = np.random.randn(n) * sqrt(2.0/n)
```

Un altro metodo per affrontare il problema della varianza è quello di impostare la matrice dei pesi a zero, ma collegando ogni neurone in modo casuale a pochi altri neuroni con pesi impostati secondo una piccola distribuzione gaussiana; questo metodo è chiamato **sparse initialization**.
Infine, una tecnica recente e molto comune, che prende il nome di **Batch Normalization**, prevede di integrare la rete con strati aggiuntivi subito dopo i livelli completamente connessi o convoluzionali, in cui gli ouptut di questi ultimi vengono normalizzati. Questa operazione permette di ottenere reti significativamente più robuste alle cattive inizializzazioni, il tutto aggiungendo solo due parametri, la media del batch e la deviazione standard del batch.

2.6.3 Regolarizzazione

Uno dei più comuni incidenti di percorso nell'addestramento di una rete neurale è quello di ottenere una rete fatta su misura per il set di addestramento, ovvero ottenere una rete che si comporti bene sul training set, ma male sui set di test; questo problema si chiama *overfitting*. Uno dei metodi per far fronte a questa situazione è chiamato **regolarizzazione**.
La tecnica più popolare prende il nome di **regolarizzazione L2**, in cui viene aggiunto un termine di penalizzazione di magnitudine quadrata alla funzione di perdita, vale a dire che per ogni peso $\boldsymbol{w}$ si aggiunge un termine $\frac{1}{2}\boldsymbol{\lambda w^2}$, dove $\boldsymbol{\lambda}$ è la forza di regolarizzazione. Questo metodo tende a penalizzare vettori di pesi con grandi fluttuazioni, prediligendo quindi valori più omogenei.

Un'altra soluzione è l'adozione della **regolarizzazione L1**, parente della versione L2 in cui il termine di penalizzazione si riconduce a $\lambda|w|$. Questa versione di regolarizzazione tende a concentrarsi sull'utilizzo delle features più importanti piuttosto che sfruttare l'intero arsenale, dove molti pesi tendono ad avvicinarsi molto allo zero. Tale metodo è da preferire qualora si sia più interessati alla selezione delle features. Si può anche tentare una combinazione di questi due tipi di regolarizzazione, riconducendosi al termine $\lambda_1|w| + \lambda_2 w^2$.
Una forma di regolarizzazione molto popolare è quella chiamata **Dropout** che va a complementare le due tecniche presentate prima. La sua realizzazione prevede di mantenere attivi solo alcuni neuroni con una probabilità p. Questo metodo risulta essere particolarmente efficace ed è un elemento di prevenzione di overfitting su numerosi problemi. Tuttavia richiede una piccola quantità di tempo superiore per la convergenza della rete, in quanto ogni addestramento viene effettuato su un'architettura "differente". In generale, è buona pratica utilizzare il metodo L2 in combinazione con dropout.

2.7 Valorizzazione del dataset

In campo medico esiste la difficile situazione per cui il materiale a disposizione sia poco, cioè spesso i dataset risultano essere troppo piccoli per poter effettuare un addestramento della rete neurale efficace, oppure può capitare che si ricada nuovamente nel problema dell'overfitting, laddove le immagini a disposizione ritraggano gli oggetti d'interesse sempre sullo stesso lato (il fianco di un'automobile per esempio); in questo caso la rete non è in grado di riconoscere correttamente lo stesso oggetto se questo è esposto su un lato differente. E' possibile tentare di aggirare il problema ottenendo ottimi risultati effettuando delle operazioni preliminari sulle immagini. Per operazioni, si intendono quelle tecniche di trasformazione delle immagini, quali la traslazione, rotazione, riflessione, oppure ridimensionamento, variazione dei colori o dell'illuminazione, rumore e distorsioni di vario tipo. E' necessario utilizzare queste tecniche in modo ponderato, ovvero scegliere con cura le operazioni da applicare: si consideri per esempio un'operazione di riflesso sull'asse orizzontale di un'immagine di un'automobile; tale procedura porta la macchina ad avere le ruote verso l'alto e ciò risulta poco utile in quanto è improbabile trovare un'automobile con questo orientamento, a meno che non si tratti di un dataset di auto incidentate.
Questo passaggio è sempre importante e va tenuto in considerazione in ogni tipo di problema, sia che si disponga di un dataset piccolo che di uno grande, poichè

questo è un momento per arricchire e valorizzare ulteriormente il proprio dataset [17].

Capitolo 3

Riconoscimento della vena giugulare dalle immagini ultrasonografiche

3.1 Modelli di reti neurali convoluzionali

Nel capitolo precedente sono state presentate quelle che sono le peculiarità di una rete neurale convoluzionale, facendo leva su quei punti critici che fanno la differenza sulla bontà di un suo addestramento. Un altro aspetto particolarmente importante è il modello utilizzato, vale a dire la struttura e le relazioni dei suoi neuroni: quanti strati possiede, il numero di neuroni di ogni strato e le relazioni di input e output. Tipicamente i neuroni di una CNN sono distribuiti tridimensionalmente all'interno di uno strato, dato che le immagini sono costruite sulle tre dimensioni di lunghezza, altezza e profondità, in cui quest'ultima ne indica i volumi di attivazione, cioè il numero di filtri. In Figura 3.1 è illustrato un semplice modello adatto ad un problema di classificazione, in cui un'immagine data in input viene "compressa" in un output monodimensionale, i cui elementi rappresentano la probabilità per cui l'oggetto rilevato nella scena appartenga a quella classe. Come mostrato nel capitolo precedente gli strati che compongono la rete neurale svolgono determinate funzioni che tramite varie combinazioni è possibile ottenere risultati diversi. Le tipologie di strati più popolari sono quelli convoluzionali, completamente connessi, RELU, Pooling.
Negli ultimi anni numerose architetture di reti neurali convoluzionali sono state proposte; alcune sono diventate vere pietre miliari, nonché punti di partenza per le generazioni successive. Vengono presentati ora alcuni dei modelli più popolari e significativi che hanno dato una svolta a questa materia.

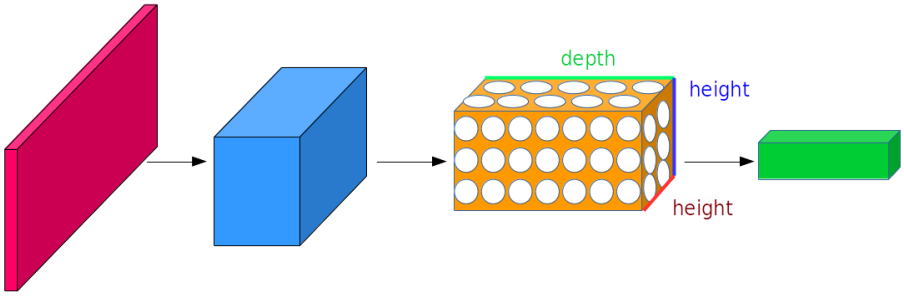


Figura 3.1: Rete Neurale convoluzionale con i neuroni disposti tridimensionalmente. Il primo strato di input è grande quanto l'immagine: lunghezza, altezza e canali RGB.

3.1.1 LeNet

Questo modello [18] è nato alla fine degli anni novanta proponendo l'obiettivo di riconoscere i numeri scritti a mano o con macchina da scrivere e presenta un'architettura semplice composta da due set di strati convoluzionali e di pooling medi, seguito da un altro strato di convoluzione , due strati completamente connessi ed un classificatore softmax:

- Primo strato: Di tipo convoluzionale, contiene 6 filtri 5x5 ed elabora un'immagine di input 32x32 in scala di grigi riducendola a 32x32x6;
- Secondo strato: Viene effettuato il pooling medio con un filtro 2x2, riducendo l'immagine a 14x14x6;
- Terzo strato: Altro strato convoluzionale con 16 filtri 5x5. In questo non tutti i filtri sono collegati ai sei precedenti, rompendo così la simmetria e mantenendo un numero di connessioni più limitato;
- Quarto strato: Ancora una volta viene effettuata un'operazione di pooling con 16 un filtri 2x2 riducendo l'output a 5x5x16;
- Quinto strato: Completamente connesso con 120 filtri 1x1 ai 400 punti dello strato precedente (di dimensione 5x5x16);
- Sesto strato: Nuovamente completamente connesso con 84 unità;
- Settimo strato: Softmax completamente connesso con 10 possibili valori, corrispondenti ai numeri da zero a nove. Eccetto questo, tutti gli strati hanno come funzione di attivazione *tanh*

Questa tipologia di rete non si presta ad elaborare immagini di grandi dimensione ed è quindi limitata dall'impossibilità di utilizzare maggiori risorse.

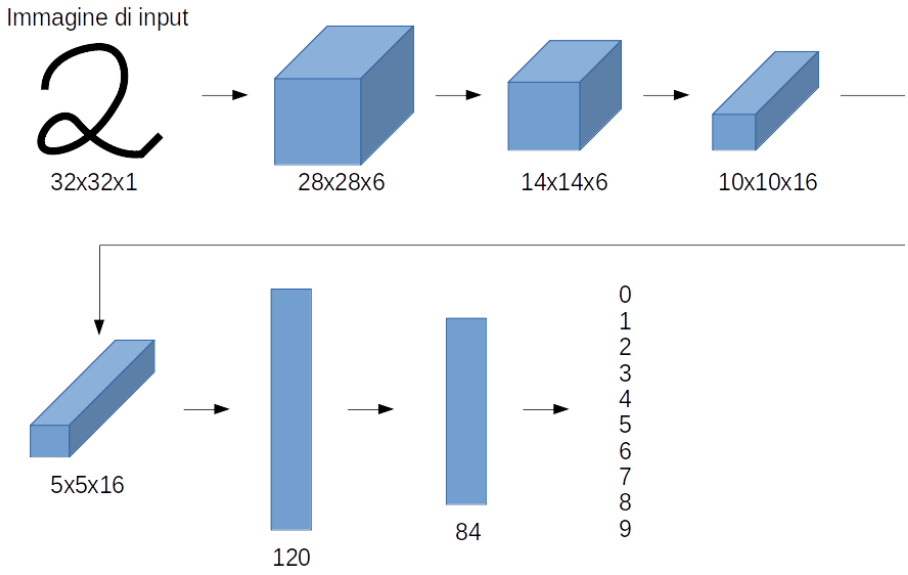


Figura 3.2: Rappresentazione volumetrica dell'architettura LeNet.

3.1.2 AlexNet

Questo modello è stato il vincitore del Large Scale Visual Recognition Challenge 2012 (ILSVRC2012), competizione dedicata ai problemi di classificazione. Questo modello si avvale delle moderne tecnologie hardware e software dedicate alla computazione, in particolare le GPU NVIDIA e CUDA, spingendo la profondità dell'architettura a valori maggiori, aumentandone anche la complessità interna delle connessioni.
Il modello originale [19] comprende principalmente otto strati, di cui i primi cinque convoluzionali e gli ultimi tre completamente connessi. Alcuni degli strati convoluzionali sono seguiti da livelli di max pooling e viene utilizzato ReLU in ognuno degli otto strati (Figura 3.3).

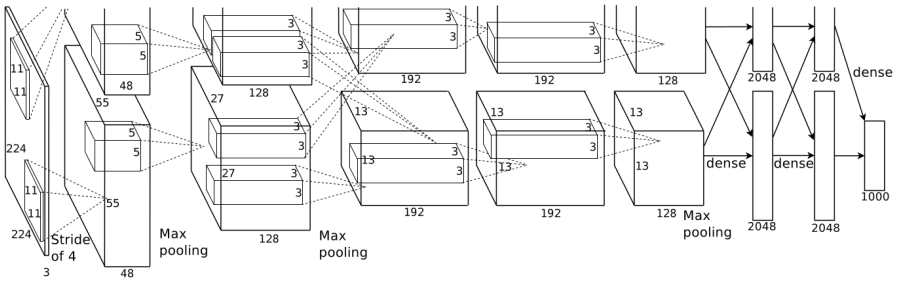


Figura 3.3: Rappresentazione volumetrica dell'architettura AlexNet.

3.1.3 GoogleLeNet

Nel 2014 Google vince la competizione ILSVRC14 presentando GoogleLeNet [20] o "Inception V1". Il tasso di errore in quella sfida lo porta vicinissimo al livello di percezione umano portando i responsabili della competizione a dare una valutazione oggettiva di quello che potesse essere il tasso d'errore della percezione umana. Questa nuova rete neurale convoluzionale trae ispirazione dalla sua antenata *LeNet*, inserendo però un nuovo "modulo" chiamato *inception* e implementando la batch normalization, distorsioni delle immagini e l'RMSprop (un'estensione della discesa stocastica del gradiente). Il modello sfrutta l'utilizzo di piccole convoluzioni, consentendo un notevole risparmio sul numero di parametri utilizzati, da sessanta milioni di AlexNet a quattro milioni, in ventidue strati convoluzionali (Figura 3.4).

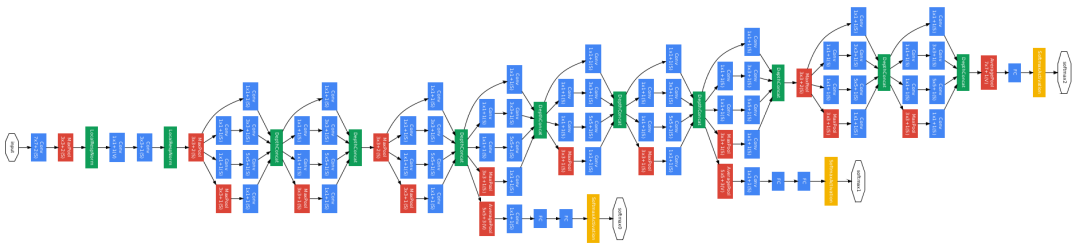


Figura 3.4: Diagramma della rete neurale convoluzionale GoogleLeNet. In blu gli strati di convoluzione, in rosso quelli di pooling, in giallo quelli di softmax, in verde resto.

3.1.4 VGGNet

Un altro modello che ha suscitato particolare interesse nel 2014 è stato quello denominato *VGGNeT* [21], composto da sedici strati convoluzionali disposti linearmente e molteplici filtri 3x3 con profondità incrementale, ma con riduzione del volume tramite gli strati di max pooling. Al termine, un classificatore softmax provvede a determinare l'output della rete. Per via della numerosità dei parametri (fino a 138 milioni) può essere difficile da contenere, ma è tuttavia una delle reti neurali più utilizzate per l'estrazione di features (Figura 3.5).

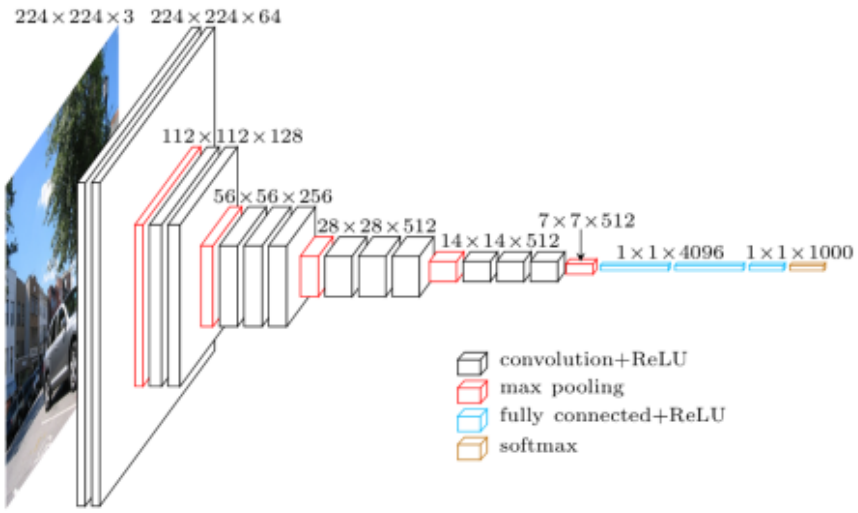


Figura 3.5: Diagramma della rete neurale convoluzionale VGGNet.

Alcuni framework per l'implementazione delle reti neurali mettono a disposizione alcune versioni di questo modello con dei pesi già inizializzati, ed unita ad un altro modello, o parte, non addestrato e permette di estrapolare numerose features evitando ore di addestramento; questo metodo è chiamatao **pre-training** o preaddestramento. La logica di questo approccio risiede nell'utilizzare la rete preaddestrata per riconoscere le features più semplici e generiche, come bordi o forme geometriche semplici (dipende dalla profondità di addestramento della rete preaddestrata) e utilizzare il proprio modello per il rilevamento di caratteristiche più "profonde" e dettagliate.

3.1.5 ResNet

Un nuovo modello è nato nella competizione ILSVRC15 con il nome Residual Neural Network [22] che spezza la normale linearità delle precedenti reti e che si

manifesta con dei ”salti” nelle connessioni tra gli strati. Questo modello introduce anche il concetto di architettura a blocchi, rendendo il modello più modulare e ricomponibile, riconducendosi all'*apprendimento residuo*, in cui, dato un blocco (uno stack di strati), il suo output è definito come l'unione del suo input con la *mappatura residua da apprendere* (le operazioni contenute nel blocco) (Figura 3.6). Questa tecnica permette di addestrare una rete neurale di ben 152 strati, mantenendo comunque una complessità inferiore a quella di VGGNet e superando il livello di percezione umana con buon margine sul dataset della competizione.

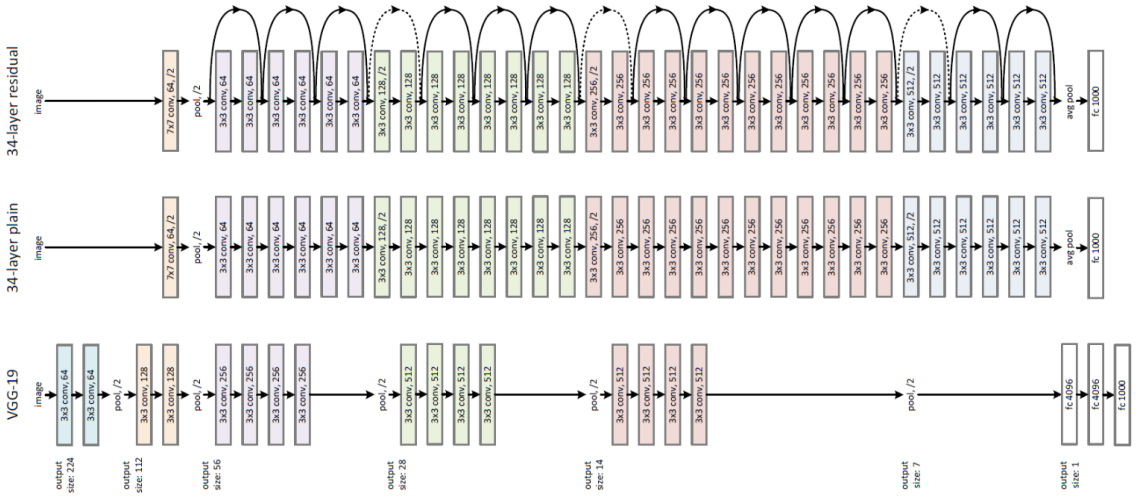


Figura 3.6: Diagramma della rete neurale convoluzionale ResNet. Da notare le connessioni che saltano alcuni blocchi.

Ad oggi (2019) alcuni dei modelli più performanti sono varianti di queste ultime due architetture presentate, variando essenzialmente in volume.

3.1.6 U-Net

Un ultimo modello da osservare è certamente U-Net [6], in quanto protagonista di questo lavoro di tesi, proposto nel 2015 come rete neurale convoluzionale per la segmentazione di immagini biomediche. L'attenzione su questa rete è stata posta sulla capacità di ottenere brillanti risultati partendo da un dataset molto esiguo.
La potenza di questo modello risiede nella concatenazione della canonica rete contraente con una espansiva, in cui gli operatori di pooling sono sostituiti da operatori di upsampling, mantenendo quindi una risoluzione dell'output più elevato. La rappresentazione grafica di questo modello assomiglia ad una U, da cui il suo nome U-Net (Figura 3.7). Per riuscire nell'obiettivo, le features più gran-

di della parte di rete in contrazione vengono combinate con quelle nella parte espansiva ed elaborate da alcuni strati convoluzionali per assemblare un output più preciso. Un'altra caratteristica peculiare della parte di upsampling è che il numero di canali presenti sono più grandi, consentendo di propagare maggiori informazioni ai livelli di risoluzione più alta.

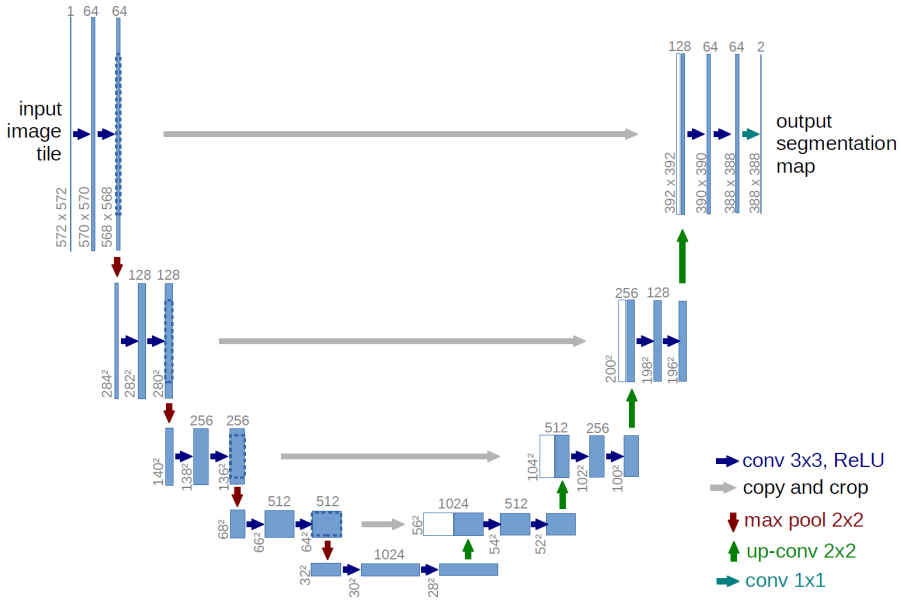


Figura 3.7: La particolare architettura di U-Net ha una forma ad U. In fase "discendente" la profondità degli strati aumenta, mentre in fase "ascendente" cala, fino ad arrivare ad una classificazione. Da notare anche la concatenazione tra le due parti.

3.2 Tensorflow e U-Net

Per questo lavoro di tesi è stato scelto di utilizzare il modello U-Net in quanto risulta essere popolare per la risoluzione di problemi legati alla segmentazione di immagini biomediche, ed in questo caso, il compito è di segmentare immagini ecografiche della sezione trasversale della vena giugulare acquisite in B-mode, isolando il vaso sanguigno dal resto dei tessuti.
Per l'implementazione si è ritenuto opportuno orientarsi su Tensorflow [23], framework open source di Google rilasciato nel 2015 appositamente per lo sviluppo più agevole di reti neurali e per software per la risoluzione di problemi legati al machine learning. E' possibile utilizzarlo sfruttando diversi strumenti hardware,

come le GPU, che consentono di effettuare computazioni numeriche ad alte prestazioni. Inoltre, tramite apposite API è utilizzabile sfruttando diversi linguaggi di programmazione come C, Java, Python, ed è fruibile su diverse piattaforme, anche mobile, al fine di rendere disponibile questo potente strumento a più persone possibile ed integrarlo ai diversi ecosistemi di software nel mondo; si basti pensare a come tanti marchi popolari lo utilizzino: eBay, Intel, Twitter e persino Coca Cola. Tensorflow si basa sulla computazione di un flusso di dati all'interno di un grafo orientato, il quale acquisisce nel tempo diversi stati e reperibili in ogni momento in base alle necessità. Parte del nome deriva dal tipo di dato che manipola, vettori multidimensionali, denominati tensori. Tutte queste caratteristiche rendono questo framework il candidato ideale per uno sviluppo di una rete neurale, in particolar modo è consigliato per chi è alle prime armi, in quanto la documentazione è ampia abbastanza da capire come utilizzarlo.
La costruzione di una rete neurale in Tensorflow avviene in modo ben strutturato, cioè si descrive la struttura della rete neurale e si definisce come deve apprendere ed immagazzinare le informazioni e infine si indica il ciclo di apprendimento. La realizzazione della rete neurale di questo lavoro di tesi è basata su un'implementazione preesistente, sviluppata da Joel Akeret [7] utilizzando Python, tramite le API di Tensorflow che lui ha utilizzato per altri scopi. L'architettura della rete costruita segue versione standard definita di U-net, lasciando all'utente flessibilità sui parametri di addestramento.

3.2.1 Implementazione U-net

La rete neurale è costruita su quattro moduli python:

- **unet.py**: modulo portante su cui tutta la rete funziona, viene costruita e addestrata;
- **layers.py**: modulo ausiliare per la realizzazione degli strati della rete;
- **util.py**: modulo accessorio contenente alcune funzione di generica utilità;
- **image_util.py**: utile modulo per la manipolazione delle immagini;

La descrizione di tali moduli viene ora effettuata in ordine diverso rispetto a quanto elencato, in modo tale da costruire il puzzle a partire dagli elementi più piccoli.

image_util.py

Dovendosi occupare della manipolazione delle immagini questo modulo si avvale dell'utilizzo delle librerie **numpy** per il calcolo vettoriale e **PIL** per la gestione delle immagini. Il modulo implementa una classe astratta *BaseDataProvider* che da la possibilità di interpretare le immagini in diversi modi. Tale classe implementa l'importante funzione per la normalizzazione dei dati, operazione quasi fondamentale per ottenere buoni risultati con le reti neurali e fornisce anche una funzione sovrascrivibile per fare operazioni di data augmentation. Infine, possiede un metodo richiamabile per ottenere un numero arbitrario di immagini e che verrà quindi usato dalla rete per ottenere i batch set di volta in volta. Questo modulo fornisce anche una classe basata su *BaseDataProvider* con le quali è già possibile caricare le immagini (insieme alle etichette, o ground truth) in modo agevole indicandone il formato immagine, il numero di classi di oggetti da segmentare e se effettuare il caricamento delle immagini in modo casuale.

util.py

Anche questo modulo utilizza le librerie per la manipolazione delle immagini *PIL* e per il calcolo vettoriale *numpy* in quanto fornisce alcune funzioni di generica utilità, come la funzione adibita alla generazione di un'immagine che affianca l'immagine di input alla ground truth e alla predizione, in modo da avere un riscontro visivo e un confronto tra i risultati (3.8). Per fare queste operazioni si rende necessaria l'utilizzo della libreria *matplotlib* che si presta in modo eccellente a questo scopo.

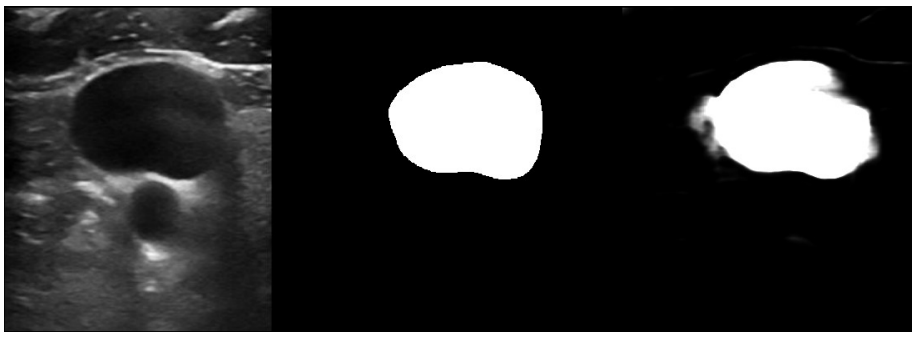


Figura 3.8: Esempio di immagine concatenata tramite il modulo util. A sinistra l'immagine di input, al centro l'etichetta di ground truth e a destra la predizione della rete neurale.

Il modulo fornisce inoltre una funzione per convertire un array in un'immagine RGB forzando la struttura dell'array e riempiendo le nuove posizioni con degli zeri. E' possibile anche fare il crop di un'immagine, ritagliarla, funzionalità usata principalmente nella funzione per la concatenazione delle immagini di input e output. Infine, è presente una funzione per il calcolo della metrica **IoU** (discussa più avanti), non originaria di questo modulo ed implementata per la valutazione delle prestazioni della rete neurale.

layers.py

Come si può dedurre dal nome di questo modulo, in questo file sono definite le funzioni dedicate alla creazione dei vari strati della rete neurale, ed è qui che si vedono i primi stralci di utilizzo del framework *tensorflow*. Qui si può osservare la funzione *weight_variable(shape, stddev)* con la quale è possibile ottenere l'insieme inizializzato dei pesi, indicando la forma della struttura che deve contenerli e una deviazione standard (di default è impostata a 0.1); Per questa inizializzazione viene usata la funzione *tensorflow.truncated_normal* che restituisce una distribuzione normale troncata.
La funzione *conv2d(x, W, keep_prob)* consente la creazione di uno strato convoluzionale con *x* i dati di input, *W* la struttura contenente i pesi e *keep_prob* la probabilità con cui viene effettuato il dropout dei neuroni. Questa procedura si avvale della funzione *tf.nn.conv2d* in cui sono impostati come parametri *x*, *W*, i valori di strides (1, 1, 1, 1 di default) che indicano di quanti pixel "spostarsi" nell'analizzare l'immagine rispetto al punto attuale e il valore di padding. A differenza della funzione del software originale, questa è stata integrata con un'operazione di **batch normalization** utilizzando la libreria *tensorflow.contrib.layers* tramite la funzione *layers.batch_norm*. Con questo arricchimento si rende la rete neurale più resistente alle cattive inizializzazioni dimostrandone l'utilità in fase di test della rete, migliorando significativamente i risultati ottenuti. Infine, questo modulo implementa le funzioni degli altri strati utilizzati, come quello di *max pooling* o alcune funzioni di utilità come quella per la concatenazione tra gli strati di compressione con quelli di upsampling.

unet.py

L'ultimo modulo che viene trattato è quello relativo alla costruzione e addestramento della rete. Le librerie utilizzate sono di vario genere, dal framework Tensorflow a numpy. Come si potrebbe intuire, il modulo contiene una classe

Unet la quale implementa la rete neurale convoluzionale relativa; il costruttore d'inizializzazione è definito come segue:

```
def __init__(self, channels=3, n_class=2, cost="cross_entropy",
                    f_threshold = 200, cost_kwargs={}, **kwargs):
```

Tale definizione prevede alcuni parametri impostati di default, ma variabili; è possibile specificare il numero di canali che possiedono le immagini, il numero di classi da identificare (nel caso di questo lavoro di tesi due: lo sfondo e il vaso sanguigno), la funzione di costo, la soglia in pixel per la metrica f-score ed eventuali impostazioni per le funzioni di costo. Le variabili vengono quindi inizializzate in base alle proprie preferenze e viene lanciata la funzione per implementare la rete neurale. Il codice di questo passaggio non viene trattato qui in quanto il modello della rete è stato descritto prima, ma è consultabile in appendice A come riferimento. Per il confronto tra la predizione e la ground truth si utilizza la funzione *pixel_wise_softmax_2* contenuta nel modulo layers e che permette di creare la mappa di probabilità di segmentazione tra la ground truth e la predizione, tramite la classificazione softmax.
la **funzione di costo** è poi definita dando la possibilità di scegliere tra la *cross entropy* ed il *dice coefficient*, la prima definita come:

```
...
loss_map = tf.nn.softmax_cross_entropy_with_logits_v2(logits=flat_logits, labels=flat_labels)
weighted_loss = tf.multiply(loss_map, weight_map)
loss = tf.reduce_mean(weighted_loss)
```

L'istruzione *tf.nn.softmax_cross_entropy_with_logits_v2* è quella incaricata di calcolare la softmax cross entropy mentre le altre due provvedono a dare la forma finale alla funzione di costo.

La seconda funzione (dice coefficient) è invece definita come:

```
...
prediction = pixel_wise_softmax_2(logits)
intersection = tf.reduce_sum(prediction * self.y)
union = eps + tf.reduce_sum(prediction) + tf.reduce_sum(self.y)
loss = -(2 * intersection / (union))
```

Come per definizione si ricavano le intersezioni e le unioni tra le predizioni e le ground truth e le si mettono in rapporto per ottenere la funzione di costo finale.
Identificate le funzioni di costo, esse vengono integrate con la regolarizzazione L2 (discusso nel capitolo 2):

```
regularizers = sum([tf.nn.l2_loss(variable) for variable in self.variables])
loss += (regularizer * regularizers)
```

Viene poi definita la funzione per la **predizione**, la quale necessita solo del riferimento ad un modello addestrato o checkpoint salvato della rete neurale e di alcuni dati (immagini) di input. La parte saliente di questa funzione è data dall'istruzione:

```
prediction = sess.run(self.predicter, feed_dict={self.x: x_test, self.y: y_dummy,
                                                 self.keep_prob: 1.})
```

Tale istruzione indica di richiamare le operazioni definite per il parametro *self.predicter*, indicate in fase di definizione del modello della rete (*pixel_wise_softmax_2*) con i parametri raggruppati nel parametro *feed_dict*.

All'interno del modulo *unet.py* è definito l'oggetto **Trainer**, la quale provvede all'addestramento della rete neurale. Per il corretto funzionamento necessità quindi di aver indicati diversi parametri, come la dimensione dei *batch*, la rete neurale stessa e l'ottimizzatore *adam* o *momentum* per influenzare il tasso di apprendimento (learning rate). L'ottimizzatore è una metodologia di del gradiente che permette di averne un controllo maggiore. Adam prende il nome da *Adaptive Momentum Estimation* ed è una variante della discesa stocastica del gradiente che prende i benefici delle due varianti AdaGrad e RMSProp, il quale influenza adattivamente il tasso di apprendimento di ogni peso, non mantenendone uno per tutti, facendo stime su un primo e un secondo 'momento' del gradiente. Indicando con w_t i pesi dell'iterazione t, con m il primo momento, con v il secondo, $\hat{\text{m}}$ e $\hat{\text{v}}$ stimatori corretti e con Q la funzione di costo, l'aggiornamento dei parametri w_{t+1} è quello dato nella Formula 3.1.

$$\begin{aligned} m_{t+1} &= \beta_1 m_t + (1-\beta_1)\,\nabla Q\,(w_t) \\ v_{t+1} &= \beta_2 v_t + (1-\beta_2)\,(\nabla Q\,(w_t))^2 \\ \hat{m} &= \frac{m_{t+1}}{1-\beta_1^{t+1}} \\ \hat{v} &= \frac{v_{t+1}}{1-\beta_2^{t+1}} \\ w_{t+1} &= w_t - \eta \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon} \end{aligned} \tag{3.1}$$

Con β_1 e β_2 parametri di controllo per il decadimento esponenziale, ϵ è un termine per la stabilità numerica e η è il tasso d'apprendimento. Questo ottimizzatore è considerato tra i più potenti ed ha acquisito forte popolarità.
Momentum invece è un termine sommato al gradiente al fine di regolarizzare la variazione dei parametri. E' dipendente dall'iterazione corrente e da quella precedente ed introduce un valore α che quantifica l'importanza della dell'iterazione passata. La Formula 3.2 lo definisce.

$$\begin{aligned} \Delta w_{t+1} &= \alpha \Delta w_t - \eta \nabla Q_i \left(w_t \right) \\ w_{t+1} &= w_t + \Delta w_t \\ w_{t+1} &= w_t - \eta \nabla Q_i \left(w_t \right) + \alpha \Delta w_{t+1} \end{aligned} \tag{3.2}$$

Nella fase d'inizializzazione del trainer, oltre alle variabili utili per il suo funzionamento vengono impostati i parametri necessari al monitoraggio dell'addestramento tramite Tensorboard, piattaforma che accompagna Tensorflow per la visualizzazione dei parametri di addestramento durante la sua esecuzione. I parametri di base monitorati sono la **perdita**, la **cross entropy** e l'**accuratezza**, mentre quelli che sono stati aggiunti durante lo studio sono le due metriche **IoU** (Intersection over Union) e l'**F-score**.
L'*Iou*, conosciuta anche come **indice di Jaccard** è una misura di similarità e diversità tra un insieme di oggetti ed è definita come il rapporto tra la dimensione dell'intersezione e la dimensione dell'unione degli insiemi degli oggetti:

$$J\left(A, B\right) = \frac{|A| \cap |B|}{|A| \cup |B|} = \frac{|A| \cap |B|}{|A| + |B| - |A| \cap |B|} \tag{3.3}$$

L'implementazione di questa metrica è consultabile in Appendice A ed è contenuto nel modulo *util.py*
L'*F-score* o **F-measure** è una misura di accuratezza del test che tiene in considerazione la 'precisione', rapporto tra il numero di veri positivi trovati e il numero di positivi ottenuti dal classificatore e il 'richiamo', rapporto tra i veri positivi e il numero totale di positivi che si sarebbero dovuti identificare:

$$F_1 = 2 \cdot \frac{p \cdot r}{p + r} \tag{3.4}$$

L'implementazione di questa metrica è lasciata alla libreria *tensorflow.contrib.metrics.f1_score*
L'oggetto *Trainer* implementa il metodo **train** con il quale si avvia l'addestramento vero e proprio della rete neurale, specificandone la durata in termini di epoche

e iterazioni. Inoltre permette d'impostare la probabilità di dropout della rete e se avviare l'addestramento su un modello precedentemente addestrato, in modo tale da arricchirlo ulteriormente. L'insieme d'istruzioni è abbastanza intuitivo ed è riportato in Appendice A, con l'unica nota di prestare attenzione alle variabili *test_x* e *test_y* che contengono i batch relativi alla fase i validazione che avviene al termine di ogni epoca, mentre le variabili *batch_x* e *batch_y* si riferiscono ai batch di training. Tutti gli altri metodi servono per la gestione dell'output e che si occupano di salvare il modello addestrato e di registrarne i checkpoint epoca dopo epoca.

Capitolo 4

Addestramento e Risultati

4.1 Dataset

I dati raccolti sono stati gentilmente forniti dall'Azienda Ospedaliero-Universitaria di Ferrara e comprendono una serie d'immagini ecografiche acquisite in B-mode di ventitre studi differenti da due cicli cardiaci ognuno per un totale di 5552 immagini più le corrispettive immagini di ground truth. Le immagini di etichetta sono state ottenute tramite un software analitico adoperato nello studio correlato a questo lavoro di tesi e sono state validate e considerate come corrette dai medici collaboratori. Il set è stato diviso in due parti: una da venti studi come set di training e tre studi per la fase di test. Il set di validazione durante la fase di addestramento viene automaticamente gestito internamente dalla rete neurale, ripartendo già preventivamente il training set in due parti. Il modello U-net è stato pensato per funzionare nelle situazioni in cui si hanno poca disponibilità di immagini e avvalendosi massicciamente alla data augmentation. In questo caso, le immagini sono risultate abbondanti, tuttavia molto simili tra loro all'interno di un singolo studio. In ogni caso, dei test usando poche immagini supportate dalla data augmentation sono stati fatti.
Di seguito si fa riferimento al set di addestramento con set_A e a quello di test con set_T

4.2 I test

Si è scelto di suddividere i test secondo due tipologie: l'utilizzo della data augmentation o meno. Entrambi casi contengono addestramenti che variano su alcuni parametri, quali la profondità della rete in termini di operazioni di sam-

pling/unsampling e di features, la lunghezza dell'addestramento in termini di iterazioni ed epoche. Di preciso, alcuni modelli addestrati e che vengono portati all'attenzione in questa tesi hanno le caratteristiche della Tabella 4.1:

Livelli U-Net	Filtri	Verification batch	Iterazioni	Epoche
4	128	16	64	40
4	64	16	64	40
4	32	16	64	40
5	128	16	64	40
5	64	16	64	40
5	32	16	64	40
5	32	32	64	100

Tabella 4.1: Tipologia di modelli addestrati e durata di addestramento

Per orientarsi meglio nella tabella è sufficiente leggerla considerando le righe ordinate per livelli e per filtri. L'ottimizzatore utilizzato è '**momentum**' con momento α impostato a 0.9 (si veda la Formula 3.2), la funzione di costo è '**cross entropy** $H(p,q)$ definita dall'equazione 4.1, con p e q distribuzioni di probabilità, rispettivamente quelle volute e quelle attuali.

$$H(p,q) = -\sum_{x} p(x) log q(x) \tag{4.1}$$

La dimensione del **kernel** è 3x3' e la probabilità di **dropout** è 0.75. L'operazione di classificazione dei pixel dell'immagine viene effettuata dello strato di output che implementa la funzione **softmax** definita dall'equazione 4.2, con **z** il vettore in ingresso e $j = 1, ..., K$ e che quindi dato il vettore **z** di K elementi, lo normalizza in una distribuzione di probabilità di K probabilità.

$$\sigma\left(\mathbf{z}\right)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{4.2}$$

L'output della rete sarà quindi un'immagine in scala di grigi in cui ogni pixel rappresenta la probabiltà che il pixel appartenga ad una classe piuttosto che un'altra. La durata di un addestramento è pari all'esecuzione del numero di iterazioni per il numero di epoche impostate. Un'iterazione corrisponde all'elaborazione di un mini batch. La macchina con la quale sono stati effettuati i test è stata gentilmente fornita dall'Università di Ferrara, accessibile mediante connessione remota ed è dotata di questo hardware:

- Processore: Intel Xeon E5 v4 Family 6 core @ 3.60GHz;
- Scheda Video: NVIDIA GeForce GTX 1080;
- RAM: 32GB;

L'ambiente di sviluppo è su un sistema Linux CentOS utilizzando Python 3.6.5 in ambiente virtuale Anaconda 3 ed è doveroso dire che i risultati ottenuti possono differire usando strumentazioni differenti, in particolar modo in base alla GPU utilizzata, poiché implementano delle politiche di ottimizzazione sui calcoli e sulla gestione della memoria che possono influire sul risultato finale.
Effettuare test con parametri superiori a quelli mostrati in tabella è stato impossibile a causa delle limitazioni dell'hardware, tuttavia la combinazione di queste specifiche ha permesso di raggiungere risultati notevoli. Gli addestramenti delle reti neurali con queste caratteristiche richiedono tempo a causa della loro complessità, nonché anche l'analisi richiede molto tempo per via della connessione su una macchina remota e i dati che sono dimensioni di diversi GB per modello. Ne consegue quindi che si sono dovute fare delle scelte sulla via da percorrere e privilegiare quindi alcuni modelli addestrati rispetto ad altri.
Gli script usati sono due e differiscono solo dall'integrazione della data augmentation che avviene a runtime prima dell'addestramento e quest'ultimo script è consultabile in appendice A. I due programmi danno la possibilità all'utente di impostare da riga di comando i parametri relativi al modello della rete neurale e dell'addestramento e dopo aver impostato TRE semi per il generatore di numeri casuali, si procede con l'alterazione delle immagini per aumentare la quantità e la qualità del dataset set_A e per ogni seme viene effettuato un addestramento. Al termine di ogni addestramento vengono rimosse le immagini generate per l'augmentation e vengono salvati il modello addestrato ed un numero impostato dall'utente di predizioni effettuate sul test set. Vengono presentati ora i vari test seguendo la tabella mostrata prima.

4.2.1 Test senza data augmentation

Le prime tre architetture presentate sono caratterizzate da un modello U-Net a quattro livelli di profondità, come da definizione del modello originale e sono diversificati tra loro nella variazione del numero di filtri usati: 128, 64, 32. Per ognuno di essi sono stati addestrati tre modelli in modo tale da non dare per assoluto un solo test e considerando fattori che generano casualità come il dropout

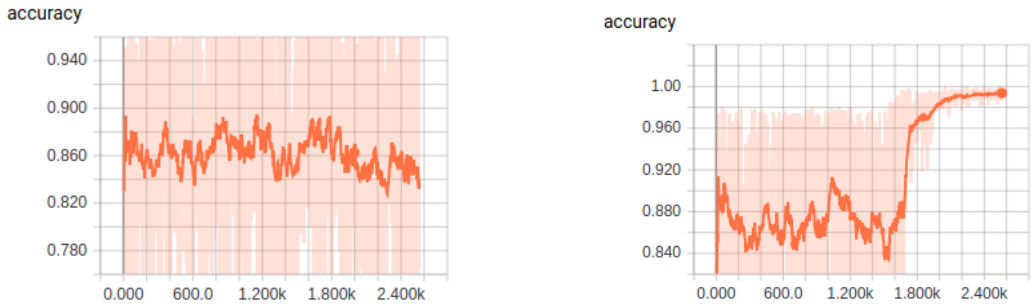


Figura 4.1: A sinistra un esempio di oscillazione dell'accuratezza. A destra un esempio in cui la rete ha un'oscillazione più debole e una maggiore accuratezza. In ascisse le iterazioni. In ordinata il valore di Accuratezza. Le immagini si riferiscono allo stesso modello di rete.

e la sequenza di immagini analizzate. I mini batch sono di dimensione 1, quindi si rientra nel contesto del batch stocastico (si veda il Capitolo 2). Non è stata utilizzata la data augmentation e i valori riportati sono quelli all'ultima iterazione dell'addestramento, fatta eccezione per alcuni valori di accuratezza, in quanto in certi test è risultata particolarmente oscillante e ne viene riportato l'intervallo entro cui rientra tale oscillazione e l'ultima iterazione dell'addestramento non risulta essere un criterio oggettivo per questa situazione particolare. Tali fluttuazioni possono essere attribuibili al raggiungimento di uno stato di stabilità in cui la rete non è ancora 'brava' a fare le predizioni e non riesce a migliorarsi e dunque con la dimensione dei mini batch a 1 tra un'iterazione e l'altra la rete fa predizioni più o meno accurate (Figura 4.1).
La discesa del gradiente utilizzata è momentum con momento 0.9 e learning rate fissato a 0.2. I dati sono riportati da Tensorboard, applicando un'operazione di 'smussatura' d'intensità 0.9 sui dati per poter interpretare meglio le oscillazioni sui dati. Si ricorda inoltre che i valori sono compresi tra 0 e 1 e che è preferibile avere valori di F-Score, IoU (Formule 3.4 e 3.3) e Accuratezza **alti** e valore di Cross Entropy **basso**. I risultati sono riassunti nella Tabella 4.2

Modello	F-score	IoU	Cross Entro-py	Accuratezza
l4 f128	0.948	0.899	0.032	0.910 - 0.930
l4 f128	0.959	0.921	0.007	0.995
l4 f128	0.975	0.951	0.005	0.996
l4 f64	0.949	0.900	0.032	0.91 - 0.94
l4 f64	0.970	0.940	0.006	0.996
l4 f64	0.949	0.900	0.027	0.900 - 0.94
l4 f32	0.949	0.900	0.032	0.91 - 0.94
l4 f32	0.970	0.940	0.006	0.996
l4 f32	0.949	0.900	0.027	0.900 - 0.940

Tabella 4.2: Addestramenti sui modelli a quattro livelli senza data augmentation.

Questi primi test mostrano come ci siano alcuni esperimenti più fruttuosi rispetto ad altri, in particolar modo risultano essere nettamente più precisi i modelli che escono dal limbo delle forti oscillazioni, ovvero quelli riportati con un valore di accuratezza fisso, arrivando a valori prossimi alla perfezione (rispetto alla ground truth di riferimento).
Prendendo come riferimento le tre immagini con relative verità di Figura 4.2 prelevate dal set set_T si può vedere un esempio di predizione su quelle immagini con i modelli migliori di questo primo insieme in Figura (4.3).

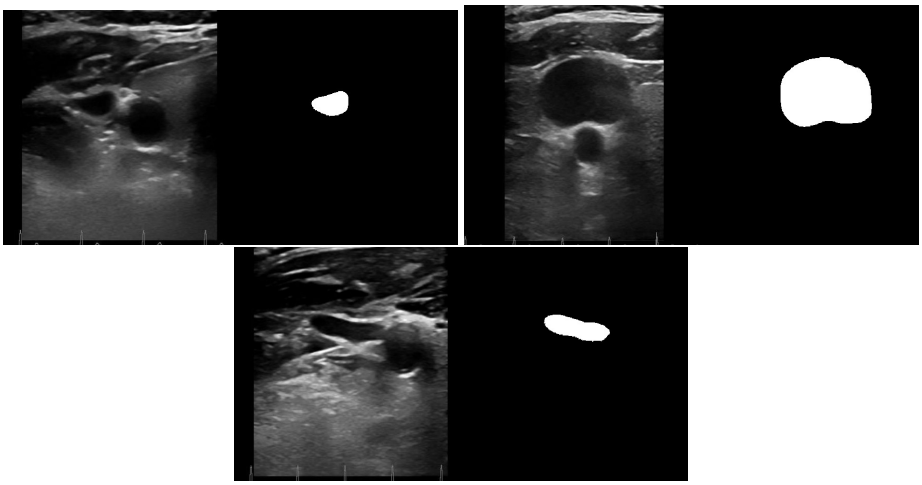


Figura 4.2: Immagini prelevate dall'inseme set_T.

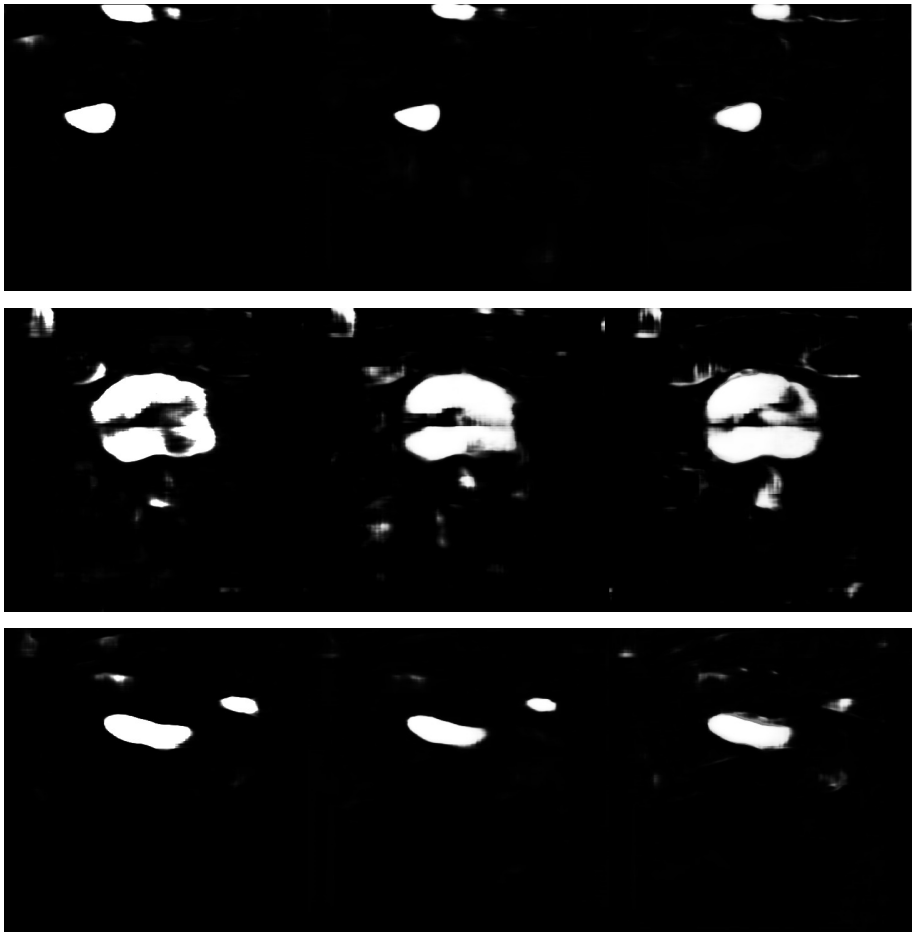


Figura 4.3: Test su immagini dell'inseme set_T. Da sinistra: 32 filtri, 64 e 128.

La prossima serie di test di Tabella 4.3 la scia degli esperimenti precedenti, ma utilizzano un'architettura U-Net a 5 livelli e con addestramento da con 40 epoche.

Modello	F-score	IoU	Cross Entro-py	Accuratezza
l5 f128	0.917	0.843	0.062	0.834 - 0.885
l5 f128	0.915	0.831	0.047	0.849 - 0.895
l5 f128	0.915	0.831	0.005	0.829 - 0.889
l5 f64	0.938	0.877	0.009	0.993
l5 f64	0.915	0.831	0.042	0.826 - 0.888
l5 f64	0.912	0.826	0.056	0.853 - 0.886
l5 f32	0.932	0.880	0.009	0.994
l5 f32	0.914	0.830	0.006	0.830 - 0.890
l5 f32	0.949	0.900	0.040	0.900 - 0.940

Tabella 4.3: Addestramenti sui modelli a cinque livelli senza data augmentation in 40 epoche.

Aver aumentato la complessità della rete neurale non sembra aver portato particolari benefici portando a molti casi stazionamento e incertezza nella rete e mantenendo una media nell'accuratezza molto bassa rispetto ai casi a 4 livelli. In particolar modo i casi a 5 livelli con 128 filtri non hanno mai mostrato segni di vantaggio nel loro impiego per questo tipo di problema. In Figura 4.4 e 4.5 è possibile vedere alcune predizioni sulle immagini di Figura 4.2 con i modelli più performanti a cinque livelli per ogni quantità di filtri, i cui risultati risultano essere più scarsi.

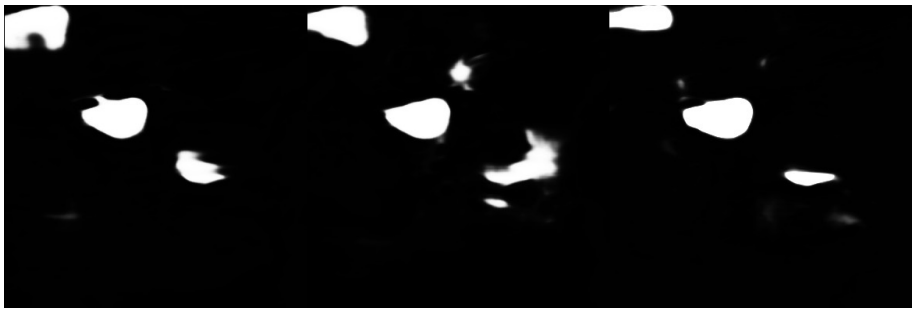


Figura 4.4: Test sulle immagini prelevate dall'inseme set_T. Da sinistra: 32, 64 e 128 filtri.

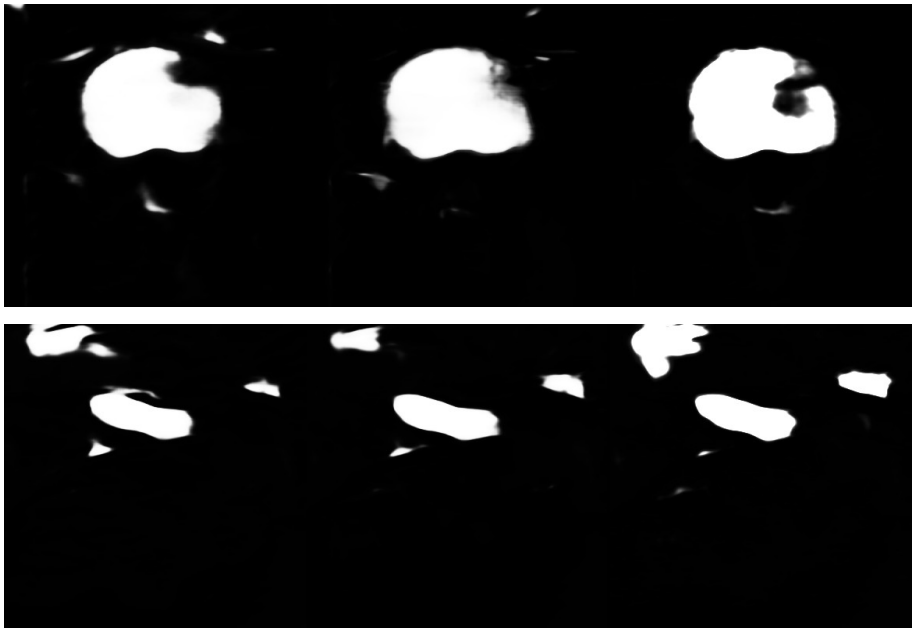


Figura 4.5: Test sulle immagini dell'inseme set_T. Da sinistra: 32, 64, 128 filtri.

Per migliorare la situazione su questi modelli si suggerisce di manipolare i parametri relativi al gradiente e al learning rate, per esempio usando come metodologia di discesa del gradiente *adam* (Equazione 3.1) per tentare di uscire da quello che potrebbe essere un minimo locale in cui la rete neurale non riesce a venirne fuori. Eventualmente beneficiare di una macchina più potente per poter fare addestramenti più lunghi. Tuttavia un tentativo aumentando le epoche di addestramento è stato fatto, portandole a 100, ottenendo i valori di Tabella 4.4:

Modello	F-score	IoU	Cross Entropy	Accuratezza
l5 f32	0.928	0.856	0.051	0.834 - 0.888
l5 f32	0.975	0.950	0.006	0.995
l5 f32	0.923	0.847	0.006	0.993

Tabella 4.4: Addestramenti su un modello a cinque livelli con 100 epoche.

Questo test porta a pensare che un tempo maggiore di elaborazione porti ad avere più possibilità di convergere a parametri migliori, senza però discostarsi dai migliori risultati ottenuti e lasciando comunque lo scettro ai modelli a quattro livelli. Avendo a disposizione questa serie di modelli può essere utile metterli alla

prova, ma avrebbe poco senso testarli tutti, in particolar modo quelli meno performanti ed è quindi preferibile mettere sotto 'stress' il modello che ha totalizzato il punteggio migliore, vale a dire il modello a quattro livelli di profondità con 128 filtri del terzo esperimento, poiché è rimasto sopra gli altri concorrenti su tutti i parametri, seppur di poco dai compagni sul podio.
Sono stati presi 1000 file dal test set e divisi in dieci parti e successivamente testati con il modello addestrato al fine di ottenere dieci valori di margine d'errore diversi. Su questi risultati si ha una media aritmetica pari a 1.03% con deviazione standard di 0.137 e limiti dell'intervallo di confidenza al 95% di 0.968 e 1.139, suggerendo quindi una stabilità complessiva sul livello di abilità del modello addestrato. Per dare un riscontro visivo di esempio su una predizione si propone la Figura 4.6.

Figura 4.6: A sinistra l'immagine di input. Al centro la ground truth. A destra la predizione.

4.2.2 Test con augmentation

Questa tipologia di test è anche il punto di partenza e di forza di U-Net, ovvero che questo modello è nato con l'intento di risolvere problemi di segmentazione delle immagini partendo con un set di dati minimo e sfruttando quindi la data augmentation per popolare l'insieme dei dati, dato che di solito il materiale a disposizione in campo medico è esiguo per questo tipo di lavori. Sono state così selezionate alcune delle immagini a disposizione, ventuno per l'esattezza, le quali durante l'esecuzione del programma vengono moltiplicate per effetto della data augmentation, realizzando per ognuna di esse tre varianti in cui vengono applicate delle operazioni di traslazione orizzontale o verticale oppure una riflessione dell'immagine. Tutte le trasformazioni hanno la stessa probabilità di essere applicate, una su due. Le traslazioni applicate spostano l'immagine di 50 pixel

per la prima immagine, 75 per la seconda, 100 per la terza.
Gli esempi mostrati di seguito riprendono le stesse tipologie di modello dei test precedenti ottenendo i valori di Tabella 4.5per i modelli a quattro livelli.

Modello	F-score	IoU	Cross Entro-py	Accuratezza
l4 f128	0.955	0.912	0.024	0.920 - 0.951
l4 f128	0.931	0.872	0.026	0.921 - 0.948
l4 f128	0.953	0.908	0.011	0.989
l4 f64	0.951	0.904	0.028	0.922 - 0.945
l4 f64	0.956	0.913	0.027	0.915 - 0.955
l4 f64	0.954	0.910	0.025	0.937 - 0.954
l4 f32	0.971	0.944	0.004	0.997
l4 f32	0.948	0.899	20.028	0.910 - 0.955
l4 f32	0.971	0.944	0.003	0.997

Tabella 4.5: Addestramenti sui modelli a quattro livelli con data augmentation in 40 epoche.

Con questi modelli è stato difficile trovare una punto di stabilità molto accurato, al punto da trovare solo due buoni candidati. Una predizione sui migliori tre modelli può essere visionato in Figura 4.7 e in Figura 4.8.

Figura 4.7: Test su immagini dell'inseme set_T. Da sinistra: 32 filtri, 64 e 128.

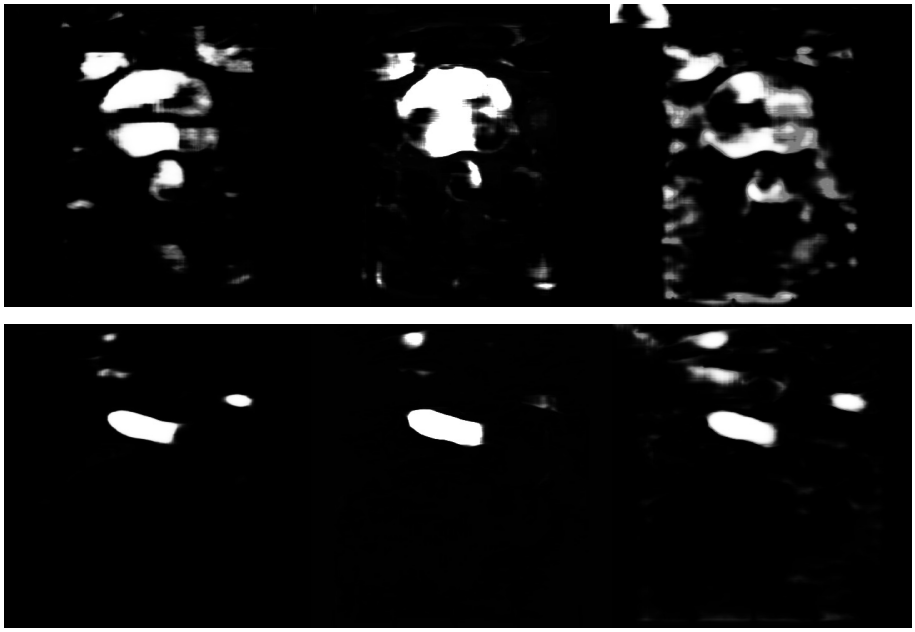


Figura 4.8: Test su immagini dell'inseme set_T. Da sinistra: 32 filtri, 64 e 128.

Sui modelli a cinque livelli invece si sono verificate le prestazioni di Tabella 4.6.

Modello	F-score	IoU	Cross Entropy	Accuratezza
l5 f128	0.925	0.852	0.042	0.850 - 0.917
l5 f128	0.921	0.855	0.046	0.868 - 0.906
l5 f128	0.928	0.857	0.042	0.850 - 0.909
l5 f64	0.922	0.846	0.043	0.851 - 0.925
l5 f64	0.928	0.859	0.045	0.856 - 0.925
l5 f64	0.955	0.913	0.007	0.994
l5 f32	0.928	0.858	0.050	0.860 - 0.921
l5 f32	0.927	0.856	0.039	0.912 - 0.845
l5 f32	0.928	0.858	0.003	0.918 - 0.860

Tabella 4.6: Addestramenti sui modelli a cinque livelli con data augmentation in 40 epoche.

Neanche questo turno di esperimenti ha dato risultati brillanti salvo qualche eccezione. Tali risultati non indicano in alcun modo che il data augmentation sia un elemento di svantaggio, anzi, per migliorare il risultato può essere conveniente aumentare il numero di casi e di trasformazioni e inoltre si suggerisce di usare

gli espedienti sul learning rate e il gradiente suggeriti nei casi precedenti. Un esempio visivo su questi modelli con i dati di test dell'insieme set_T è possibile osservarlo in Figura 4.9

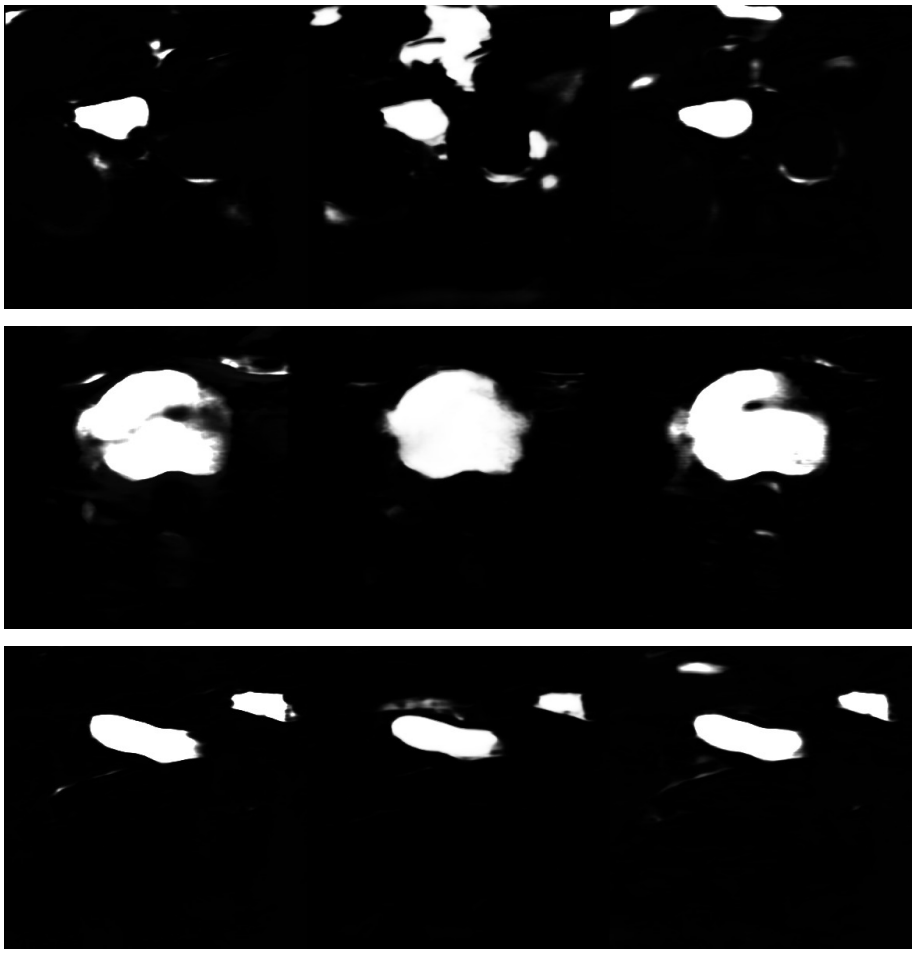


Figura 4.9: Test su immagini dell'inseme set_T. Da sinistra: 32 filtri, 64 e 128.

Anche in questa situazione è stato fatto un test con il modello migliore, ovvero quello a quattro livelli a trentadue filtri del terzo run. Ripetendo il medesimo esperimento sui dati di test set_T si è ottenuta una media aritmetica di 1.70% con deviazione standard 0.748 e dei limiti di intervallo di confidenza al 95% da 1.235 a 2.164. Seppure sia un risultato peggiore il seguente esempio grafico di Figura 4.10 mostra comunque un buon esito complessivo.
Entrambe queste tipologie soffrono di un grave problema: raggiungono una convergenza stabile ad elevata accuratezza in pochi casi. Ciò ha condotto a riflettere

Figura 4.10: A sinistra l'immagine di input. Al centro la ground truth. A destra la predizione.

sulle cause di questo comportamento, portando alla conclusione che la rete potrebbe essere incastrata in un minimo locale, raggiungendo uno stato di grande incertezza. Pertanto l'utilizzo di una discesa del gradiente più morbida è sembrata la cosa più sensata e quindi qualche esperimento con l'ottimizzatore momentum a momento 0.2 è stato effettuato sui modelli con cinque livelli, ottenendo i seguenti di Tabella 4.7.

Test a momento basso senza data augmentation

Modello	F-score	IoU	Cross Entropy	Accuratezza
l5 f64	0.967	0.936	0.015	0.989
l5 f64	0.927	0.854	0.045	0.800 - 0.920
l5 f64	0.910	0.820	0.007	0.820 - 0.911
l5 f32	0.967	0.936	0.015	0.989
l5 f32	0.918	0.854	0.045	0.815 - 0.908
l5 f32	0.910	0.822	0.057	0.810 - 0.905

Tabella 4.7: Valutazione dei modelli a 5 livelli con momento a 0.2.

Purtroppo non è stato possibile reperire i test con 128 filtri a causa di imprevisti tecnici. Questi risultati non hanno dato il risultato sperato, lasciando spazio all'incertezza e ad una scarsa convergenza dei modelli e pertanto testarne le capacità risulta inutile e tanto vale proseguire valutando dei modelli con l'ausilio della data augmentation visionabili in Tabella 4.8.

Test a momento basso con data augmentation

Modello	F-score	IoU	Cross Entropy	Accuratezza
l5 f128	0.962	0.926	0.015	0.990
l5 f128	0.967	0.936	0.012	0.993
l5 f128	0.969	0.940	0.014	0.991
l5 f64	0.956	0.915	0.012	0.992
l5 f64	0.965	0.932	0.013	0.992
l5 f32	0.920	0.833	0.540	0.800 - 0.921
l5 f32	0.919	0.830	0.052	0.819 - 0.912
l5 f32	0.919	0.930	0.63	0.820 - 0.910

Tabella 4.8: Valutazione dei modelli a 5 livelli con momento a 0.2 e data augmentation.

In questa serie di test gli addestramenti risultano più stabili senza superare in termini assoluti i modelli precedenti. Tuttavia bisogna tenere in considerazione che in questi casi, con momento a 0.2 i tempi di convergenza sono stati raggiunti prima, arrivando anche a 500 iterazioni di differenza. Questo suggerisce che "giocare" con gradiente è importante per l'ottimizzazione della rete.
Infine, in questi esperimenti un evidente problema è quello dell'overfitting, ovvero che questi modelli risultano molto bravi nel riconoscere sulle proprie immagini di training, raggiungendo picchi di quasi perfezione, ma non lo sono altrettanto sui dati di test, seppur senza troppi margini di differenza in alcuni casi.

4.3 Pregi e difetti

Tutti questi addestramenti mostrano come raggiungere l'obiettivo prefissatosi all'inizio sia attuabile, ma non senza incontrare delle difficoltà. Portare questa rete neurale ad una convergenza in questo tipo di problema è possibile ma difficile senza le adeguate strumentazioni e le giuste configurazioni. In particolar modo occorre indagare sulle cause di tale instabilità e studiare in modo più approfondito i parametri che guidano il gradiente ed utilizzare qualche espediente. Una possibile causa dell'instabilità della rete è attribuibile ad un learning rate eccessivo, mentre le cause dell'overfitting sono da ricercare nel data set che seppur di grande dimensione, richiede maggiore variabilità.

Ad ogni modo, raggiunto questo risultato è possibile selezionare le immagini più convincenti e ottenere l'area della sezione trasversale della vena giugulare applicando un'operazione di soglia, in modo tale da eliminare i valori d'incertezza che causano solo rumore sul risultato finale e facendo emergere la parte interessata. Inoltre, è possibile identificare la modalità di acquisizione dell'originale e determinare un'indicazione generica da dare all'operatore dell'ecografo su come acquisire l'immagine. Un ulteriore analisi si potrebbe fare ritagliando preliminarmente l'immagine originale dal contorno, area in cui vi sono le informazioni sul paziente ed i suoi parametri, che potrebbe portare ulteriore confusione in fase di addestramento.

Conclusioni

Sebbene i progressi fatti in questi ultimi anni in campo medico nei confronti delle patologie a carico del sistema nervoso siano notevoli, per molte malattie non esistono ancora informazioni certe sulle terapie più efficaci, sulla progressione delle manifestazioni cliniche nel tempo e su molti altri temi.
Tra queste malattie si annovera la sclerosi multipla, una malattia in alcuni casi particolarmente invalidante, anche in considerazione del fatto che esordisce in età giovanile e determina un netto peggioramento della qualità di vita dei pazienti affetti.
La tesi focalizza l'attenzione sulla CCSVI, l'insufficienza venosa cronica cerebrospinale, che, secondo gli studi condotti dal Dottor Paolo Zamboni, medico chirurgo dell'Università di Ferrara, potrebbe avere importanti correlazioni con la sclerosi multipla. L'associazione tra le due malattie è ancora oggetto di studio, ma è emerso che la formazione di stenosi e le malformazioni genetiche o acquisite delle vene cerebrospinali causano un insufficiente afflusso di sangue al sistema nervoso centrale che, col passare del tempo, lede le capacità psicomotorie del paziente, il quale manifesta sintomi sovrapponibili a quelli delle persone affette da sclerosi multipla.
Lo strumento che più frequentemente si usa per indagare questo genere di problematiche è l'ecografo, che permette di valutare in modo non invasivo nel tempo l'andamento della malattia e la sua progressione, inoltre può essere riutilizzato ogni qualvolta sia necessario, non essendo dannoso per la salute come ad esempio i raggi X, e il suo uso non prevede costi di nessun genere, se non quelli legati all'operatore.
Per questo motivo l'obiettivo della tesi è quello di proporre un software in grado di analizzare i vasi sanguigni tramite lo studio di immagini ultrasonografiche, in particolare si è scelto di esaminare le immagini relative alla vena giugulare interna, un vaso principale facilmente localizzabile e che costituisce dunque un ottimo elemento per l'osservazione di disfuzioni del sistema cardiovascolare del distretto anatomico superiore, le quali naturalmente possono riflettere una situazione

sistemica alterata.
Il software proposto basa il suo funzionamento sul machine learning, più precisamente sulle reti neurali, adottando una variante del modello tipico impiegata nell'analisi delle immagini, che si chiama rete neurale convoluzionale o CNN. Questa variante è conosciuta per praticità anche con il nome di U-net, il cui nome deriva dalla forma grafica ad "U" che viene assunta dal modello e il cui funzionamento consiste nel comprimere l'immagine in ingresso al sistema per estrapolarne solo alcune caratteristiche e poi decomprimere l'immagine in uscita.
Lo scopo ultimo di questo lavoro è di realizzare una rete neurale in grado di riconoscere le sezioni trasversali della vena giugulare interna dalle immagini ultrasonografiche acquisite in B-Mode, al fine di ricavarne poi un giugulogramma in grado di valutare la dilatazione del vaso tra una pulsazione e l'altra e l'estrapolazione di altri importanti dati e parametri.
In conclusione è opportuno valutare il raggiungimento di tale obiettivo: in effetti questo software è in grado di evidenziare la sezione trasversale della vena giugulare interna, ma per quanto riguarda i dati di test il margine di errore risulta considerevole, mentre tale margine si assottiglia fino a valori decisamente accettabili per quanto riguarda i dati di addestramento, con i quali il software può raggiungere un'accuratezza che raggiunge il 99%, che è maggiore delle prestazioni generalmente considerate per gli ecografisti.
Tuttavia, come già evidenziato, sui dati di test la rete non è così efficiente, ma esistono diversi espedienti per migliorarla:

- Tuning: questo termine si usa per indicare la procedura di affinamento dei dei parametri che contribuisce in modo determinante a raggiungere un migliore risultato.

- Reset del learning rate: questo procedimento implica la possibilità di resettare il learning rate dopo averne raggiunto una certa soglia. Tale manovra è necessaria per spostare il gradiente della rete neurale da un eventuale minimo locale, aumentando possibilmente il livello di convergenza. Tale tecnica è descritta in []

- Utilizzo di una rete pre-addestrata: è possibile impiegare una rete addestrata in modo rudimentale cioè in grado di riconoscere le figure più semplici piuttosto che delle caratteristiche degli oggetti e lasciare la parte di più sofisticata in grado di riconoscere caratteristiche più peculiari all'attuale rete. Esempi di questo tipo vedono l'utilizzo di versioni pre-addestrate delle reti VGG come TernausNet;

- Eliminazione dell'overfitting: l'overfitting è quel fenomeno per il quale la rete risulta funzionare meglio con i dati di addestramento e peggio con i dati di test. Insistendo con le varie tecniche di prevenzione è probabile che si riesca ad ottenere risultati migliori e meno specifici.

Il motivo per il quale non sono state indagate in maniera esaustiva queste possibilità è dovuto ad un limite oggettivo intrinseco alla strumentazione a disposizione per questo lavoro di tesi: infatti sia la macchina personale che quella a disposizione dell'Università (sebbene più potente di quella domestica) non sono in grado di portare a termine in tempi brevi gli addestramenti richiesti. Per maggiore chiarezza bisogna considerare che alle volte può essere avviato un addestramento che viene completato in più di quattro ore. E' opportuno anche pensare che usando parametri più raffinati il tempo di addestramento si dilati ulteriormente rispetto ai tempi già estremamente lunghi visti fino ad ora e lasciando spazio al caso.
In conclusione è verosimile ritenere che con una strumentazione più avanzata e un lavoro eseguito da un team di esperti sarebbe possibile ottenere risultati eccellenti in questo settore, producendo un software le cui prestazioni superino quelle dell'operatore umano e così riducano il rischio di errore a cui ogni medico può, suo malgrado, essere esposto, inoltre potrebbero ridursi notevolmente i tempi di esaminazione di un'ecografia e ciò comporterebbe, in ultima analisi, una riduzione dei tempi di attesa per la prenotazione di esami ecografici che è un obiettivo prioritario per la Sanità Pubblica e per lo Stato.

Appendice A

A.1 Creazione della rete neurale U-Net

```
def create_conv_net(x, keep_prob, channels, n_class, layers=3, features_root=16, filter_size=3,
pool_size=2, summaries=True):

"""
Creates a new convolutional unet for the given parametrization.

:param x: input tensor, shape [?,nx,ny,channels]
:param keep_prob: dropout probability tensor
:param channels: number of channels in the input image
:param n_class: number of output labels
:param layers: number of layers in the net
:param features_root: number of features in the first layer
:param filter_size: size of the convolution filter
:param pool_size: size of the max pooling operation
:param summaries: Flag if summaries should be created
"""

logging.info("Layers {layers}, features {features}, filter size {filter_size}x{filter_size},
pool size: {pool_size}x{pool_size}".format(layers=layers, features=features_root,
filter_size=filter_size,pool_size=pool_size))

# Placeholder for the input image
nx = tf.shape(x)[1]
ny = tf.shape(x)[2]
x_image = tf.reshape(x, tf.stack([-1, nx, ny, channels]))
in_node = x_image
batch_size = tf.shape(x_image)[0]

weights = []
biases = []
convs = []
pools = OrderedDict()
deconv = OrderedDict()
dw_h_convs = OrderedDict()
```

```
up_h_convs = OrderedDict()

in_size = 1000
size = in_size
# down layers
for layer in range(0, layers):
features = 2 ** layer * features_root
stddev = np.sqrt(2 / (filter_size ** 2 * features))
if layer == 0:
w1 = weight_variable([filter_size, filter_size, channels, features], stddev)
else:
w1 = weight_variable([filter_size, filter_size, features // 2, features], stddev)

w2 = weight_variable([filter_size, filter_size, features, features], stddev)
b1 = bias_variable([features])
b2 = bias_variable([features])

conv1 = conv2d(in_node, w1, keep_prob)
tmp_h_conv = tf.nn.relu(conv1 + b1)
conv2 = conv2d(tmp_h_conv, w2, keep_prob)
dw_h_convs[layer] = tf.nn.relu(conv2 + b2)

weights.append((w1, w2))
biases.append((b1, b2))
convs.append((conv1, conv2))

size -= 4
if layer < layers - 1:
pools[layer] = max_pool(dw_h_convs[layer], pool_size)
in_node = pools[layer]
size /= 2

in_node = dw_h_convs[layers - 1]

# up layers
for layer in range(layers - 2, -1, -1):
features = 2 ** (layer + 1) * features_root
stddev = np.sqrt(2 / (filter_size ** 2 * features))

wd = weight_variable_devonc([pool_size, pool_size, features // 2, features], stddev)
bd = bias_variable([features // 2])
h_deconv = tf.nn.relu(deconv2d(in_node, wd, pool_size) + bd)
h_deconv_concat = crop_and_concat(dw_h_convs[layer], h_deconv)
deconv[layer] = h_deconv_concat

w1 = weight_variable([filter_size, filter_size, features, features // 2], stddev)
w2 = weight_variable([filter_size, filter_size, features // 2, features // 2], stddev)
b1 = bias_variable([features // 2])
b2 = bias_variable([features // 2])

```

```
conv1 = conv2d(h_deconv_concat, w1, keep_prob)
h_conv = tf.nn.relu(conv1 + b1)
conv2 = conv2d(h_conv, w2, keep_prob)
in_node = tf.nn.relu(conv2 + b2)
up_h_convs[layer] = in_node

weights.append((w1, w2))
biases.append((b1, b2))
convs.append((conv1, conv2))

size *= 2
size -= 4

# Output Map
weight = weight_variable([1, 1, features_root, n_class], stddev)
bias = bias_variable([n_class])
conv = conv2d(in_node, weight, tf.constant(1.0))
output_map = tf.nn.relu(conv + bias)
up_h_convs["out"] = output_map

if summaries:
for i, (c1, c2) in enumerate(convs):
tf.summary.image('summary_conv_%02d_01' % i, get_image_summary(c1))
tf.summary.image('summary_conv_%02d_02' % i, get_image_summary(c2))

for k in pools.keys():
tf.summary.image('summary_pool_%02d' % k, get_image_summary(pools[k]))

for k in deconv.keys():
tf.summary.image('summary_deconv_concat_%02d' % k, get_image_summary(deconv[k]))

for k in dw_h_convs.keys():
tf.summary.histogram("dw_convolution_%02d" % k + '/activations', dw_h_convs[k])

for k in up_h_convs.keys():
tf.summary.histogram("up_convolution_%s" % k + '/activations', up_h_convs[k])

variables = []
for w1, w2 in weights:
variables.append(w1)
variables.append(w2)

for b1, b2 in biases:
variables.append(b1)
variables.append(b2)

return output_map, variables, int(in_size - size)
```

A.2 Algoritmo per il calcolo della metrica IoU

```
    def get_iou_vector(A, B):
    """
    Calculate IoU between labels and predictions to retrieve a metric

    :param A: ground truth
    :param B: predictions
    :return: the mean IoU over the batch
    """
    #thresholding to 0-1
    """
    nmax = 1
    nmin = 0
    for i in range(A.shape[0]):
    l = A[i]
    p = B[i]
    A[i] = (l - np.amin(l)) * (nmax - nmin) / (np.amax(l) - np.amin(l)) + nmin
    B[i] = (p - np.amin(p)) * (nmax - nmin) / (np.amax(p) - np.amin(p)) + nmin
    """
    batch_size = A.shape[0]
    metric = 0.0
    for batch in range(batch_size):
    t, p = A[batch], B[batch]
    true = np.sum(t)
    pred = np.sum(p)

    if true == 0:
    metric += (pred == 0)
    continue
```

A.3 Algoritmo per la gestione dell'addestramento

```
    def train(self, data_provider, output_path, training_iters=10, epochs=100, dropout=0.75,
    display_step=1, restore=False, write_graph=False, prediction_path='prediction'):
    """
    Lauches the training process

    :param data_provider: callable returning training and verification data
    :param output_path: path where to store checkpoints
    :param training_iters: number of training mini batch iteration
    :param epochs: number of epochs
    :param dropout: dropout probability
    :param display_step: number of steps till outputting stats
    :param restore: Flag if previous model should be restored
    :param write_graph: Flag if the computation graph should be written as protobuf file
    to the output path
```

```
    :param prediction_path: path where to save predictions on each epoch
    """
    save_path = os.path.join(output_path, "model.ckpt")
    if epochs == 0:
        return save_path

    init = self._initialize(training_iters, output_path, restore, prediction_path)
    init_l = tf.local_variables_initializer()

    with tf.Session() as sess:
        if write_graph:
            tf.train.write_graph(sess.graph_def, output_path, "graph.pb", False)

        sess.run(init)
        sess.run(tf.global_variables_initializer())
        sess.run(init_l)

        if restore:
            ckpt = tf.train.get_checkpoint_state(output_path)
            if ckpt and ckpt.model_checkpoint_path:
                self.net.restore(sess, ckpt.model_checkpoint_path)

        test_x, test_y = data_provider(self.verification_batch_size)

        #normalization
        test_x /= np.std(test_x)
        test_y /= np.std(test_y)

        pred_shape = self.store_prediction(sess, test_x, test_y, "_init")

        summary_writer = tf.summary.FileWriter(output_path, graph=sess.graph)
        logging.info("Start optimization")

        avg_gradients = None
        for epoch in range(epochs):
            total_loss = 0
            for step in range((epoch * training_iters), ((epoch + 1) * training_iters)):
                batch_x, batch_y = data_provider(self.batch_size)

                #normalization
                test_x /= np.std(test_x)
                test_y /= np.std(test_y)

                # Run optimization op (backprop)
                _, loss, lr, gradients = sess.run(
                (self.optimizer, self.net.cost, self.learning_rate_node, self.net.gradients_node),
                feed_dict={self.net.x: batch_x,
                           self.net.y: util.crop_to_shape(batch_y, pred_shape),
                           self.net.keep_prob: dropout})
```

```
    if self.net.summaries and self.norm_grads:
    avg_gradients = _update_avg_gradients(avg_gradients, gradients, step)
    norm_gradients = [np.linalg.norm(gradient) for gradient in avg_gradients]
    self.norm_gradients_node.assign(norm_gradients).eval()

    if step % display_step == 0:
    self.output_minibatch_stats(sess, summary_writer, step, batch_x,
    util.crop_to_shape(batch_y, pred_shape))

    total_loss += loss

    self.output_epoch_stats(epoch, total_loss, training_iters, lr)
    self.store_prediction(sess, test_x, test_y, "epoch_%s" % epoch)

    save_path = self.net.save(sess, save_path)
    logging.info("Optimization Finished!")

    return save_path
```

A.4 Script per l'addestramento con data augmentation

```
    import os, sys, getopt
    os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
    from tf_unet import unet, util, image_util
    import random
    import numpy as np
    import pathlib
    from scipy.misc import imsave
    from PIL import Image
    import augmenter
    import os, shutil
    from time import sleep



    h = 'tf_jugular_unet.py [opt]\n\nValidation files must be in the "validation" folder in the
    script root directory\n\noptions:\n\n\t-h:\thelp.\n\t-l:\tnumber of layers. Default is 5.
    \n\n\t-f:\tnumber of features root. Default is 32.\n\n\t-F:\tfilter size. Default is 3.
    \n\n\t-c:\tnumber of channels. Default is 1 (greyscale).
    \n\n\t-n:\tnumber of classes. Default is 2.\n\n\t-w:\tcost value. Default is 0.001.
    \n\n\t-y:\toptimizer (adam or momentum). Default is momentum with 0.9.
    \n\n\t-s:\toptimizer value. Default for momentum is 0.9.
    \n\n\t-i:\tinput data. Default is ../train/*.tif.\n\n\t-t:\tnumber of training iterations.
    Default is 32.\n\n\t-e:\tnumber of epochs. Default is 20.
    \n\n\t-p:\tnumber of dropout probability in the interval 0-1. Default is 0.75.
    \n\n\t-o:\tdirectory where to place the output prediction.
```

```
26  ␣␣␣␣Default␣is␣./␣.\n\nend␣of␣help.\n'

28      def main(argv):
        # −h to help
30      layers = 5 # −l
        features_root = 32 # −f
32      filter_size = 3 # −F
        channels = 1 # −c
34      n_class = 2 # −n
        verification_batch = 4 # −v
36      cost_value = 0.001 # −w
        optimizer = 'momentum' # −y
38      optimizer_value = 0.9 # −s
        data = './train/*' # −i
40      training_iters = 32 # −t
        epochs = 20 # −e
42      dropout = 0.75 # −p
        output = './' # −o
44      data_sfx = ".tif" # −x
        mask_sfx = "_mask.tif" # −m
46      trained_path = "./trained/" # −O
        f_threshold = 200 # −T
48      augmentdata = '/home/mcanella/jugU−net/train−for−augmentation/*' # −g

50      try:
        opts, args = getopt.getopt(argv,"hl:f:F:c:n:v:w:y:s:i:t:e:p:o:x:m:O:T:g:")
52      except getopt.GetoptError:
        print(h)
54      sys.exit(2)
        for opt, arg in opts:
56      if opt == '−h':
        print (h)
58      sys.exit()
        elif opt == '−l':
60      layers = int(arg)
        elif opt == '−f':
62      features_root = int(arg)
        elif opt == '−F':
64      filter_size = int(arg)
        elif opt == '−c':
66      channels = int(arg)
        elif opt == '−n':
68      n_class = int(arg)
        elif opt == '−v':
70      verification_batch_size = int(arg)
        elif opt == '−w':
72      cost_value = float(arg)
        elif opt == '−y':
74      if(arg == 'momentum' or arg == 'adam'):
        optimizer = arg
```

```
    else:
    print(h)
    sys.exit(2)
    elif opt == '-s':
    optimizer_value = float(arg)
    elif opt == '-i':
    data = arg
    elif opt == '-t':
    training_iters = int(arg)
    elif opt == '-e':
    epochs = int(arg)
    elif opt == '-p':
    dropout = float(arg)
    elif opt == '-o':
    output = arg
    elif opt == '-x':
    data_sfx = arg
    elif opt == '-m':
    print("we_had_a_mask" + arg)
    mask_sfx = arg
    elif opt == '-O':
    trained_path = arg
    elif opt == '-T':
    f_threshold = int(arg)
    elif opt == '-g':
    augmentdata = arg

    seeder = np.zeros(3)
    seeder[0] = 20032014
    seeder[1] = 2021988
    seeder[2] = 21111991

    for i in range(seeder.shape[0]):

    np.random.seed(seeder[i])
    random.seed(seeder[i])
    # IMAGE AUGMENTATION
    print("Image_augmentation_with_seed_"+str(seeder[i]))

    # define the path
    dir = pathlib.Path(augmentdata)

    # define the pattern
    pattern = "*_mask.png"

    l = int(len([name for name in os.listdir(dir) if os.path.isfile(os.path.join(dir, name))]))
    printProgressBar(0, l, prefix = 'Progress:', suffix = 'Complete', length = 50)
    j = 0
    for file in dir.glob(pattern):
    fname = str(file).replace("_mask", "")
```

```
126     #print(fname)

128     img, label, orig, origlab = augmenter.data_augmentation(fname, file, 75, seeder[i])
        imsave("./tmp/"+fname.replace(augmentdata, ""), orig)
130     imsave("./tmp/"+fname.replace(augmentdata,"").replace(".png", "_mask.png"), origlab)
        imsave("./tmp/"+fname.replace(augmentdata,"").replace(".png", "_aug75.png"), img)
132     imsave("./tmp/"+str(file).replace(augmentdata, "").replace("_mask.png",
        "_aug75_mask.png"), label)
134
        img, label, orig, origlab = augmenter.data_augmentation(fname, file, 50, seeder[i])
136     imsave("./tmp/"+fname.replace(augmentdata, ""), orig)
        imsave("./tmp/"+fname.replace(augmentdata,"").replace(".png", "_mask.png"), origlab)
138     imsave("./tmp/"+fname.replace(augmentdata, "").replace(".png", "_aug50.png"), img)
        imsave("./tmp/"+str(file).replace(augmentdata, "").replace("_mask.png",
140     "_aug50_mask.png"), label)

142     img, label, orig, origlab = augmenter.data_augmentation(fname, file, 25, seeder[i])
        imsave("./tmp/"+fname.replace(augmentdata, "").replace(".png", "_mask.png"), origlab)
144     imsave("./tmp/"+fname.replace(augmentdata, ""), orig)
        imsave("./tmp/"+fname.replace(augmentdata, "").replace(".png", "_aug25.png"), img)
146     imsave("./tmp/"+str(file).replace(augmentdata, "").replace("_mask.png",
        "_aug25_mask.png"), label)
148
        sleep(0.1)
150     # Update Progress Bar
        printProgressBar(j+1, l, prefix = 'Progress:', suffix = 'Complete', length = 50)
152     j+=1


154


156
        #preparing data loading
158     data_provider = image_util.ImageDataProvider(data, data_suffix = data_sfx,
        mask_suffix = mask_sfx)
160
        #setup & training
162     net = unet.Unet(layers=layers, features_root=features_root, filter_size=filter_size,
        channels=channels, n_class=n_class, f_threshold=f_threshold,
164     cost_kwargs=dict(regularizer=cost_value))
        if(optimizer == 'momentum'):
166     trainer = unet.Trainer(net, verification_batch_size = verification_batch_size,
        optimizer=optimizer, opt_kwargs=dict(momentum=optimizer_value))
168     elif(optimizer == 'adam'):
        trainer = unet.Trainer(net, verification_batch_size = verification_batch_size,
170     optimizer=optimizer, opt_kwargs=dict(learning_rate=optimizer_value))
        trained_path = trained_path + "augemented-"+ str(i)+"/"
172     path = trainer.train(data_provider, trained_path+str(i)+"/", training_iters=training_iters,
        epochs=epochs, dropout=dropout)
174
        #verification
```

```
    data_provider = image_util.ImageDataProvider("./validation/*.tif")
    for j in range(100):
    x_test, y_test = data_provider(1)
    prediction = net.predict(path, x_test)
    # prediction[prediction <= 0.5] = 0
    # prediction[prediction > 0.5] = 1
    unet.error_rate(prediction, util.crop_to_shape(y_test, prediction.shape))
    img = util.combine_img_prediction(x_test, y_test, prediction)
    util.save_image(img, output+str(i)+"/prediction"+str(j)+".jpg")

    # delete augmented files
    folder = './tmp/'
    for the_file in os.listdir(folder):
    file_path = os.path.join(folder, the_file)
    try:
    if os.path.isfile(file_path):
    os.unlink(file_path)
    # elif os.path.isdir(file_path): shutil.rmtree(file_path)
    except Exception as e:
    print(e)

    def printProgressBar (iteration, total, prefix = '', suffix = '', decimals = 1,
    length = 100, fill = '|||'):
    """
    Call in a loop to create terminal progress bar
    @params:
    iteration   - Required  : current iteration (Int)
    total       - Required  : total iterations (Int)
    prefix      - Optional  : prefix string (Str)
    suffix      - Optional  : suffix string (Str)
    decimals    - Optional  : positive number of decimals in percent complete (Int)
    length      - Optional  : character length of bar (Int)
    fill        - Optional  : bar fill character (Str)
    """
    percent = ("{0:." + str(decimals) + "f}").format(100 * (iteration / float(total)))
    filledLength = int(length * iteration // total)
    bar = fill * filledLength + '-' * (length - filledLength)
    print('\r%s |%s| %s%% %s' % (prefix, bar, percent, suffix), end = '\r')
    # Print New Line on Complete
    if iteration == total:
    print()

    if __name__ == "__main__":
    main(sys.argv[1:])
```

Bibliografia

[1] CCSVI http://www.ccsvi-sm.org/

[2] L. Testut e A. Latarjet, "Anatomia Umana" - Vol.II, Edizione Unione tipografico-editrice torinese, 1972

[3] P. Hoskins, A. Thrush, K. Martin e T. Whittingam, "Diagnostic Ultrasound: Physics and Equipment", Cambridge University Press, 2003

[4] James H. Schwartz- Eric R. Kandel- Thomas M. Jessell, "Principi di neuroscienze" - Terza edizione, Casa Editrice Ambrosiana, 2003

[5] Paolo Pazzaglia, "Clinica Neurologica" - Società Editrice Esculapio - Settima Edizione, 2010

[6] Olaf Ronneberger, Philipp Fischer, and Thomas Brox, "U-Net: Convolutional Networks for BiomedicalImage Segmentation" - University of Freiburg, Germany - arXiv:1505.04597v1 [cs.CV] 18 May 2015

[7] Joël Akereta, Chihway Changa, Aurelien Lucchib, Alexandre Refregiera, "Radio frequency interference mitigation using deep convolutional neural networks" - arXiv:1609.09077v2 [astro-ph.IM] 13 Jan 2017

[8] Sergey Ioffe, Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training byReducing Internal Covariate Shift" - arXiv:1502.03167 [cs.LG] 2 Mar 2015

[9] P. Zamboni, R. Galeotti, E. Menegatti, A.M. Malagoni, G. Tacconi, S. Dall'Ara, I. Bartolomei and F. Salvi., "Chronic cerebrospinal venous insufficiency in patients with multiple sclerosis" - J. Neurol. Neurosurg. Psychiatry 2009

[10] Andrej Karpathy, Stanford University course CS231n, https://cs231n.github.io/

[11] Freund, Y.; Schapire, "Large margin classification using the perceptron algorithm" - Machine Learning. 37 (3): 277–296. doi:10.1023/A:1007662407062 1999

[12] Database MNIST, http://yann.lecun.com/exdb/mnist/

[13] Ian Goodfellow and Yoshua Bengio, Aaron Courville, "Deep Learning" - MIT Press, http://www.deeplearningbook.org, 2016

[14] Michael A. Nielsen, "Neural Networks and Deep Learning" - Determination Press - http://neuralnetworksanddeeplearning.com/, 2015

[15] Convoluzione - https://en.wikipedia.org/wiki/Convolution

[16] Convolutional Neural Network - https://en.wikipedia.org/wiki/Convolutional_neural_netw

[17] Alex Hernández-García, Peter König, "Do deep nets really need weight decay and dropout?" - arXiv:1802.07042 12 Jul 2018

[18] Pubblicazioni su LeNet - http://yann.lecun.com/exdb/lenet/

[19] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks" - https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[20] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, "Going Deeper with Convolutions" - arXiv:1409.4842 17 Sep 2014

[21] Karen Simonyan, Andrew Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition" - arXiv:1409.1556 10 Apr 2015

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, "Deep Residual Learning for Image Recognition" - arXiv:1512.03385 10 Dec 2015

[23] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever,

Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems" - 2015. Software available from tensorflow.org

Ringraziamenti

Il mio più profondo ringraziamento al mio relatore Prof.ssa Elena Loli Piccolomini e al mio correlatore Dott. Giovanni Di Domenico che mi hanno guidato, accompagnato e aiutato a raggiungere questo traguardo con grande disponibilità ed infinita pazienza.

Grazie a Martina, senza la quale raggiungere la fine di questo percorso e concludere questo lavoro sarebbe stato infinitamente più difficile se non impossibile. Grazie per essere sempre stata accanto a me, io lo sarò per te.

Grazie ai miei genitori che ci hanno sempre creduto e hanno continuato a sostenermi. Grazie alla mia famiglia che aspettava da tempo questo traguardo insieme a me.

Grazie a Luca F. con cui condivido tutti gli studi fatti e con cui ho trascorso le più esilaranti giornate a Bologna in questi anni.

Grazie ai miei amici più stretti che con l'umorismo e l'ironia hanno dato il loro contributo, poiché anche loro sono sempre stati fiduciosi nei miei confronti.

Grazie alla mia grande famiglia Atletica Estense, con cui condivido tutti pomeriggi e che mi da ossigeno in quei momenti della giornata.

Grazie agli amici di Widegroup che seppur nelle fasi finali di questo percorso mi hanno dato sin dal primo momento un fortissimo sostegno e una grande disponibilità per portare a termine il lavoro.

Grazie a tutti voi di cuore.