



UNIVERSITÀ
POLITECNICA
DELLE MARCHE

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA E
DELL'AUTOMAZIONE

**Classificazione di istotipi di cancro
polmonare direttamente dalla TAC: Verso
un approccio end-to-end**

**Lung cancer Histotypes classification directly from CT scans: Towards
an end-to-end approach**

Relatore: Chiar.mo
Prof. Aldo Franco Dragoni

Tesi di laurea di:
Sara Abbonizio

A.A. 2019/2020

UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE
Via Brezze Bianche – 60131 Ancona (AN), Italy

Ringraziamenti

Questa tesi è stata la conclusione di un lungo percorso che non sarei mai riuscita ad affrontare senza l'aiuto di tutte le persone che mi hanno supportato. Innanzitutto vorrei ringraziare il Prof. Aldo Franco Dragoni per avermi offerto l'opportunità di lavorare a questa tesi, e Selene Tomassini e Nicola Falcionelli per tutto l'aiuto, i consigli, la disponibilità e il supporto che mi hanno dato in questi mesi difficili, e tutto il gruppo dell'AIRTLab in generale. Non posso poi non ringraziare anche mia sorella Chiara e i miei genitori che erano sempre pronti ad ascoltarmi quando incontravo qualche problema o avevo qualche dubbio; senza di loro, non sarei mai arrivata fino a qui. Quindi grazie anche a loro per tutte le chiacchierate, le rassicurazioni, le tisane e le raccomandazioni di staccare ogni tanto e rilassarmi. Infine, grazie anche a tutti i miei amici, in particolare quelli per cui sono solo "Sis" che mi hanno sempre tenuto compagnia e fatto ridere in questi mesi, fosse anche solo via chat o chiamata vocale.

Ancona, Settembre 2020

Sara Abbonizio

Abstract

In questa tesi siamo andati ad esaminare la possibilità di utilizzare due diversi tipi di reti neurali per il riconoscimento di due diverse tipologie di cancro al polmone. Il capitolo 1 offre una prima introduzione all'argomento, con riferimenti a studi passati su argomenti simili e le motivazioni che hanno portato allo sviluppo di questa tesi. Il capitolo 2 introduce le nozioni teoriche di base sui concetti di intelligenza artificiale ed apprendimento; successivamente il capitolo 3 entra un po' più nel dettaglio illustrando il funzionamento anche dal punto di vista matematico delle due tipologie di reti utilizzate. Il capitolo 4 illustra le metriche utilizzate per valutare l'accuratezza e l'affidabilità dei risultati ottenuti dalla rete. Il capitolo 5 invece illustra cosa sono le Tomografie assiali computerizzate e i dati su cui siamo andati a lavorare, mentre il capitolo 6 il procedimento di preprocessing effettuato per ottenere dati adatti ad essere dati in input alle reti. Infine, il capitolo 7 illustra nel dettaglio tutti gli esperimenti effettuati e il capitolo 8 propone alcuni possibili sviluppi futuri del lavoro effettuato.

Indice

1	Introduzione	1
2	Intelligenza artificiale e apprendimento automatico	7
2.1	Teoria dell'apprendimento	9
3	Reti neurali artificiali	11
3.1	Reti neurali convoluzionali	16
3.1.1	Reti neurali convoluzionali 3D	19
3.1.2	Reti neurali ConvLSTM	21
4	Valutazione dell'apprendimento	25
4.1	La matrice di confusione	25
4.2	Accuratezza	27
4.3	Precisione e recupero	27
4.4	F1 score	28
4.5	ROC e AUC	29
4.5.1	Curve ROC	31
4.5.2	AUC	31
5	Tomografia assiale computerizzata	35
5.1	Tomografia Computerizzata	35
5.2	Scala di Hounsfield	37
6	Standard DICOM e preprocessing dell'immagine	39
6.1	Preprocessing dell'immagine	40

7	Esperimenti effettuati	47
7.1	Il progetto Jupyter	47
7.1.1	Google Colab	47
7.2	Linguaggi e librerie utilizzate	48
7.3	Dataset utilizzato	51
7.4	Rete C3DKeras	52
7.5	Rete ConvLSTM	54
7.6	Suddivisione degli esperimenti	56
7.7	Risultati ottenuti	63
7.8	Esperimenti sul modulo Jetson TX2	67
7.8.1	Il modulo NVIDIA Jetson TX2	67
7.8.2	Il Jetson Developer Kit	67
7.8.3	Configurazione della Jetson TX2	67
7.8.4	SDK Manager	68
7.8.5	Installazione TensorFlow e Keras	69
7.8.6	Ambiente Jupyter e Colab	70
7.8.7	Test effettuati	70
7.8.8	Risultati	72
8	Conclusioni e sviluppi futuri	73

Capitolo 1

Introduzione

Nel corso degli anni termini come "apprendimento automatico", denominato anche "machine learning"(ML) e "reti neurali" hanno acquisito sempre più importanza nell'ambito della ricerca scientifica. Le relative tecniche sviluppatesi di ML non sono infatti ormai relegate al solo campo dell'informatica e dell'intelligenza artificiale, ma hanno trovato applicazioni negli ambiti più disparati e in molti altri campi della ricerca scientifica. Alcuni esempi di applicazioni rese possibili solo grazie al ML sono ad esempio i software di riconoscimento vocale, i sistemi di guida automatica dei veicoli senza pilota, riconoscimento di immagini nell'ambito della computer vision e molti altri.

Tuttavia, una delle applicazioni più importanti del ML è indubbiamente quella in ambito clinico [1]. Requisito fondamentale del ML è infatti la necessità di avere a disposizione una grande mole di dati da analizzare, dati che vengono prodotti in ogni caso quotidianamente in ambito medico attraverso procedure diagnostiche di vario tipo. Oltre che far quindi utilizzare questi dati unicamente alla figura del medico in sé in fase di diagnosi del paziente, è ormai sempre più diffusa la pratica di utilizzare questi stessi dati come input per il ML, andando potenzialmente a creare software che possono affiancarsi all'operatore umano ed aiutarlo ad ottenere diagnosi più precise e soprattutto più rapide. La tempestività della diagnosi è spesso infatti fondamentale soprattutto nell'ambito dell'individuazione di svariate tipologie di tumori, per cui la percentuale di sopravvivenza del paziente aumenta sensibilmente quanto più rapidamente il tumore viene individuato.

Una delle patologie in particolare su cui gli studi si focalizzano maggiormente è l'individuazione di tumore ai polmoni.

Capitolo 1 Introduzione

In Italia e nel resto dei paesi industrializzati il tumore al polmone è una delle prime cause di morte in assoluto, e per l'Italia in particolare è la prima causa di morte per tumore tra gli uomini e terza causa di morte per tumore tra le donne.

Le ultime stime AIRTUM (Associazione italiana registri tumori) [2] descrivono 42.500 diagnosi di tumore del polmone solo nel 2019, di cui 29.500 negli uomini e 13.000 nelle donne. Queste diagnosi rappresentano il 15% di tutte le diagnosi di tumore negli uomini e del 12% nelle donne.

Secondo altre stime ricavate dagli ultimi dati disponibili, nel corso della propria vita mediamente un uomo su 11 e una donna su 39 possono sviluppare un tumore al polmone, mentre sempre un uomo su 11 e una donna su 45 rischiano di morire per causa diretta della patologia [2].

Il tumore del polmone può svilupparsi a partire dalle cellule che costituiscono i bronchi, bronchioli e alveoli, e può arrivare a formare una massa tumorale che può ostruire il passaggio dell'aria o provocare emorragie polmonari e bronchiali. Lo stesso polmone può inoltre diventare sede di metastasi di altri tumori che si estendono poi ad altri organi colpiti (un esempio è il tumore alla mammella).

Il tumore al polmone può suddividersi in carcinoma a cellule piccole e non piccole. Quest'ultimo è il più diffuso (85% dei casi di tumore al polmone) e può a sua volta suddividersi in adenocarcinoma e carcinoma a cellule squamose, per cui la ricerca nel ML non si limita alla sola individuazione della massa tumorale nel paziente, ma anche alla possibilità di riuscire a discernere le diverse tipologie.

In questa tesi si sono andate ad esaminare due tipologie di tumore al polmone: l'*adenocarcinoma polmonare* e il *carcinoma a cellule squamose*.

L'adenocarcinoma polmonare è una tipologia di tumore al polmone caratterizzato da cellule di grandi dimensioni; l'estrema eterogeneità istologica dell'adenocarcinoma polmonare ha portato alla sua suddivisione in due macro categorie, l'adenocarcinoma non invasivo(o minimamente invasivo) che a sua volta si suddivide in adenocarcinoma in situ del polmone(o carcinoma bronchioalveolare) ed adenocarcinoma minimamente invasivo, e l'adenocarcinoma invasivo diviso in vari sottotipi come l'adenocarcinoma a predominanza solida e l'adenocarcinoma invasivo mucinoso[3].

Il carcinoma polmonare a cellule squamose è invece una neoplasia maligna invasiva che ha origine dall'epitelio bronchiale, e costituisce circa il 30% di tutti i casi di

carcinoma polmonare [4].

Al giorno d'oggi la diagnosi di un tumore al polmone si effettua principalmente in due fasi: la prima è quella di effettuare una tomografia computerizzata o TC (CT in inglese), che permette di valutare la presenza o meno di un possibile tumore, ma non di riconoscere la tipologia. La tomografia è poi seguita dalla necessità di effettuare una biopsia, che consiste nel prelievo di una porzione di tessuto tumorale. Questa operazione è per ovvi motivi estremamente invasiva e non è completamente priva di rischi per il paziente, anche se non porta alla diagnosi di un tumore.

Una individuazione e diagnosi rapida della presenza di tumore al polmone è quindi essenziale alla sopravvivenza del paziente. Nel corso degli anni il miglioramento continuo nel campo della Tomografia Assiale Computerizzata (TAC) ha messo a disposizione dei medici che devono occuparsi della diagnostica un numero sempre crescente di dati. Le immagini TAC prodotte al giorno d'oggi sono aumentate rispetto a solo qualche decade fa non solo in numero ma anche in risoluzione e qualità, il che richiede uno sforzo e consumo di tempo a disposizione dei radiologi notevole. Un radiologo, inoltre, è sempre pronò a commettere errori e ad individuare falsi positivi a causa di stress, stanchezza o semplice errore umano. Per questo motivo da ormai quasi due decenni si stanno effettuando studi sullo sviluppo e miglioramento della diagnosi assistita dal computer o CAD (Computer Aided Diagnosis), in particolare nell'ambito delle reti neurali. La CAD è una disciplina vastissima e nell'ambito della diagnosi di tumore al polmone si può suddividere a sua volta in altre sotto-discipline a seconda del particolare problema affrontato.

Una prima problematica nello sviluppo della CAD è quella dell'individuazione dei noduli del polmone. Zhao et al (2018) [5] hanno proposto un modello di Convolutional Neural Network (CNN) per la classificazione di noduli in maligni (presenza di cancro) e benigni (assenza di cancro) denominata "Agile CNN" che raggiunge una precisione dell'82% con l'uso di soli due layer convoluzionali invece del numero maggiore richiesto da altre reti più profonde.

Wang et al (2020) [6] propongono una rete che utilizza un modello di estrazione feature multi percorso che combina le feature estratte dai primi layer convoluzionali (globali) con quelle dei layer più profondi (locali) in modo da preservare l'informazione contenuta in entrambi. Inoltre, per differenziare tra più tipologie di noduli

differenti a seconda della loro grandezza, la rete proposta implementa multipli filtri di grandezze variabili invece di un filtro a grandezza definita. Perez et al (2020) [7] hanno sviluppato un modello per l'individuazione di cancro al polmone che utilizza multiple CNN a 12 strati. La prima CNN effettua una riduzione dei falsi positivi e la classificazione di noduli su immagini pre-processate con l'applicazione di filtri e tecniche di estrazione del polmone. Le cinque tipologie di noduli più probabili generate dalla prima CNN vengono date in input ad un sistema di 5 CNN a 12 strati che condividono dei layer FC, ognuna addestrata nel riconoscimento di una tipologia di nodulo in particolare.

Molti altri studi sono stati effettuati nel campo della segmentazione e pre-processing automatico delle immagini. Una delle maggiori difficoltà che si incontrano nell'ambito del ML e in particolare del transfer learning è l'adattamento dei dati a disposizione al formato che la rete si aspetta in input a causa della sua struttura intrinseca, o della rimozione del rumore presente in un'immagine.

Il pre-processing di una immagine viene effettuato solitamente attraverso varie tecniche di elaborazione digitale delle immagini come il tresholding, wavelet e l'active countourning, ma vari studi sono stati effettuati per l'implementazione di una segmentazione automatica.

Shaziya et al (2019) [8] hanno compilato una panoramica completa dei vari metodi di segmentazione del polmone più utilizzati al giorno d'oggi, a partire da quelli convenzionali come il tresholding alle applicazioni di machine learning come l'utilizzo di support vector machine (SVM), alberi decisionali e k-nearest neighbour e illustrando infine i metodi del ML con reti neurali convoluzionali fully connected, Region-CCN e anche una implementazione di U-Net. Della rete U-Net si sono occupati anche Kunpang et al (2019) [9], implementando una struttura formata da un insieme di encoder e decoder; l'encoder prende in input l'immagine originale e produce una feature map a bassa risoluzione, dopodiché il decoder effettua l'upsampling per recuperare l'informazione contenuta nell'immagine di partenza.

Gerard et al (2020) [10] propongono una CNN che combina modelli ad alta e bassa risoluzione per la segmentazione di immagini di polmoni lesionati di svariate specie di mammiferi oltre all'essere umano, combinando immagini ottenute da dataset di pazienti umani, ovis, porcini e canini. Hu et al (2020) [11] hanno applicato la rete

Mask-R-CNN alla segmentazione di immagini del polmone ottenendo una precisione del 97%.

Come visto anche da Perez et al (2020) [7], di particolare interesse negli ultimi tempi è l'uso delle reti neurali convoluzionali a tre dimensioni.

Lima et al (2020) [12] esaminano diverse architetture di reti neurali convoluzionali 3D per la classificazione di noduli nel polmone in maligni e benigni, ottenendo una accuratezza del 90%.

Scopo di questa tesi è andare ad esaminare la possibilità di addestrare una rete neurale non al mero riconoscimento della presenza di tumore al polmone, bensì all'identificazione della sua tipologia, task che attualmente solo il corrispettivo referto istologico riesce a fare, evitando così i rischi collegati alla biopsia e riducendo notevolmente i tempi necessari ad ottenere una diagnosi certa aumentando la percentuale di sopravvivenza del paziente.

In particolare, si vuole vedere se questo è possibile attraverso l'implementazione di una rete end-to-end e l'utilizzo del transfer learning, che permette di addestrare una rete ad eseguire un compito specifico a partire da un'altra rete già addestrata in precedenza, riducendo le risorse richieste in termini di costo computazionale dell'hardware, di tempo e di quantità di dati necessari all'apprendimento.

Infine, come obiettivo finale, si è voluta andare ad esaminare la possibilità di effettuare predizioni di risultati di immagini TAC su un modulo Jetson TX2, concentrandosi principalmente sulle tempistiche richieste ad effettuare predizioni per un singolo paziente ed esaminando i risultati ottenuti per stabilire la possibile implementazione di un sistema di analisi e predizione delle TAC in tempo reale.

Capitolo 2

Intelligenza artificiale e apprendimento automatico

E' bene iniziare questa tesi con una piccola introduzione su cosa si intende esattamente per intelligenza artificiale e apprendimento automatico. Innanzitutto, secondo la definizione data dall'ingegnere Marco Somalvico [13]:

“L'intelligenza artificiale è una disciplina appartenente all'informatica che studia i fondamenti teorici, le metodologie e le tecniche che consentono la progettazione di sistemi hardware e sistemi di programmi software capaci di fornire all'elaboratore elettronico prestazioni che, ad un osservatore comune, sembrerebbero essere di pertinenza esclusiva dell'intelligenza umana”

Un sistema intelligente quindi agisce seguendo un comportamento simile a quello umano, o, in termini più generici, secondo un comportamento razionale basato su quattro principi fondamentali:

- Pensa umanamente: il processo con cui il sistema arriva alla soluzione ricalca quello umano;
- Agisce umanamente: il risultato del suo operato deve essere indistinguibile dalla stessa operazione svolta da un essere umano;
- Pensa razionalmente: il processo opera secondo un procedimento formale che segue le leggi della logica;
- Agisce razionalmente: il processo con cui il sistema intelligente risolve il problema è quello per cui è in grado di ottenere il miglior risultato.

Nel corso degli anni l'intelligenza artificiale ha portato a diversi dibattiti tra scienziati e filosofi per via di svariati problemi di natura sia etica che pratica che pone il suo

sviluppo. Nel 2017, a seguito di un convegno di esperti del settore promosso dal Future of Live Institute sono stati redatti i Principi di Asilomar [14], un vademecum con 23 principi per affrontare le problematiche etiche, sociali, culturali e militari dell'intelligenza artificiale.

La grande complessità intrinseca dell'intelligenza artificiale ha portato negli anni alla suddivisione in svariate sotto discipline, delle quali il ML è stata fin dalla sua nascita una delle discipline di maggior importanza, insieme ad altri come la rappresentazione della conoscenza o la pianificazione.

Le due discipline sono sempre proseguite di pari passo: già negli anni '50 svariati ricercatori, in particolare Marvin Minsky, Arthur Samuel e Frank Rosenblatt, tentarono di definire dei metodi formali attraverso i quali una macchina potesse apprendere dai dati che aveva a disposizione, arrivando a definire il primissimo modello di rete neurale formato da perceptroni.

Verso la metà degli anni '50, si iniziò a creare il distacco tra l'intelligenza artificiale e il ML, che venne abbandonato per svariati anni a seguito della pubblicazione nel '69 dell'opera *"Perceptrons: an introduction to computational geometry"* di Marvin Minsky e Seymour A. Papert, che descriveva alcune delle possibili limitazioni dei perceptroni e delle reti neurali. L'intelligenza artificiale invece proseguì i suoi studi sull'approccio knowledge-based o basato sulla conoscenza, che ha portato alla nascita della programmazione logica induttiva.

La ricerca sul ML è tuttavia rinata nel corso degli anni '80 e '90, con la riscoperta della back-propagation e continuando il suo sviluppo verso metodi e modelli che utilizzano principi della statistica e teoria della probabilità. L'obiettivo del ML è stato quindi spostato dal creare una vera intelligenza artificiale al risolvere problemi di natura pratica.

E' necessario menzionare il contributo fondamentale che ha dato al ML l'avvento di Internet, che ha facilitato enormemente la raccolta, distribuzione e rappresentazione dei dati, da sempre grande problema dell'approccio probabilistico all'apprendimento che lo aveva portato ad essere abbandonato per diverso tempo.

2.1 Teoria dell'apprendimento

L'obiettivo principale del ML è che una macchina sia in grado di generalizzare dalla propria esperienza, ovvero è in grado di seguire dei ragionamenti detti induttivi. Aristotele definì per la prima volta i termini con cui vengono indicate le tre modalità di ragionamento che l'essere umano può seguire, indicandole come deduzione, induzione e abduzione.

La deduzione (dal latino *de ducere*, letteralmente “condurre da”) indica il tipo di ragionamento per cui si arriva ad una conclusione da premesse più generiche basate su postulati e principi primi. In altre parole, con la deduzione si parte da leggi universali per arrivare ad una conclusione particolare.

L'induzione (dal latino *in-ducere*, letteralmente “portare fuori”) è invece il ragionamento inverso, ovvero quello per cui a partire da singoli casi particolari si cerca di arrivare a stabilire delle regole generali.

La differenza sostanziale tra deduzione ed induzione è quindi che la deduzione porta a risultati veri se parte da principi veri. La regola generale prodotta dall'induzione invece non è ovviamente sempre vera, ma è applicabile con un determinato livello di probabilità. Una macchina che apprende autonomamente è quindi una macchina che è in grado di eseguire il compito per cui è stata addestrata su esempi nuovi che non ha mai visto, dopo aver fatto esperienza su un insieme di dati di apprendimento.

Proprio per questo è di fondamentale importanza la rappresentazione e raccolta dei dati di apprendimento. Si immagini ad esempio di voler addestrare una macchina a riconoscere foto di cani e foto di gatti: idealmente, per avere una macchina con accuratezza perfetta, ovvero in grado di identificare correttamente un animale dall'altro nel 100% dei casi, le si dovrebbero fornire dati che rappresentano ogni input possibile, nel nostro esempio ogni immagine possibile di un cane o di un gatto.

Ciò non è fisicamente possibile per ovvie ragioni, quindi è necessario scegliere dati che siano rappresentativi della realtà nel modo migliore possibile.

Il ML si può a sua volta suddividere in tre categorie o paradigmi di apprendimento [15]:

- **Apprendimento supervisionato:** al modello da addestrare vengono forniti degli esempi sotto forma di input ed i rispettivi output desiderati. L'obiet-

tivo per il modello sarà estrarre una regola generale per cui è in grado di associare all'input l'output corretto. L'apprendimento supervisionato è quindi maggiormente utilizzato per risolvere problemi di classificazione o regressione.

- **Apprendimento non supervisionato:** al modello vengono forniti solo gli input senza nessuna etichettatura, lasciando al modello il compito di trovare una struttura nei dati e raggrupparli in categorie, una operazione nota con il nome di clustering.
- **Apprendimento per rinforzo:** il modello interagisce con un ambiente dinamico nel quale deve raggiungere un determinato obiettivo. Ad ogni azione effettuata dal modello è accompagnato un valore numerico detto “ricompensa”, che ha lo scopo di incoraggiare i comportamenti corretti al raggiungimento dell'obiettivo. Un apprendimento di questo tipo implica la presenza di un agente, ovvero il modello deve essere in grado di percepire l'ambiente circostante e la reazione che ogni sua possibile azione pone sull'ambiente, in modo da arrivare a individuare la sequenza di azioni migliori al raggiungimento del suo scopo. Un esempio classico di apprendimento per rinforzo è l'addestrare un modello a risolvere un determinato gioco come gli scacchi.

Capitolo 3

Reti neurali artificiali

Nel paragrafo precedente abbiamo spesso parlato dei diversi modi in cui un modello può essere in grado di apprendere, ma non abbiamo illustrato come è esattamente composto questo modello. In questo paragrafo e nei successivi si illustreranno in modo più approfondito le sue componenti principali, specificando i principi matematici su cui il modello è basato.

Innanzitutto, perché si parla di “rete neurale artificiale”? La risposta si trova sempre nei primi anni di studi dell’intelligenza artificiale: una delle idee alla base dell’intelligenza artificiale è quella di riprodurre il funzionamento di un cervello umano, composto appunto da una cosiddetta rete neurale formata da un numero incredibilmente alto di neuroni fittamente interconnessi.

Un neurone si può considerare formato da tre parti:

- Soma: il corpo principale del neurone;
- Assone: una linea di uscita dal neurone che si dirama in migliaia di rami;
- Dendrite: una linea di entrata nel neurone che riceve segnali di ingresso da altri neuroni attraverso le sinapsi.

Operazione fondamentale è che il singolo neurone è in grado di eseguire una “somma pesata” di tutti i suoi segnali in ingresso; se il valore risultante supera una determinata soglia il neurone si attiva e viene prodotto un potenziale di azione che viene trasportato all’assone per venire a sua volta trasmesso ad altri neuroni e così via. Se il valore di soglia invece non viene superato, il neurone rimane inattivo.

Questo principio viene ripreso ipotizzando di costruire un neurone artificiale: nel 1943 W.S. McCulloch e Walter Pitts nel loro lavoro “*A logical calculus of the ideas immanent in nervous activity*” schematizzano un combinatore lineare a soglia, che

prende dati binari multipli in entrata e restituisce un singolo dato binario in uscita. Connettendone un numero opportuno, si poteva calcolare il risultato di semplici operazioni booleane.

Questo combinatore era strutturalmente simile al neurone, ma non era in grado di apprendere, capacità invece acquisita come già accennato dal perceptrone e il primo schema di rete neurale introdotto da Frank Rosenblatt. Questo primo algoritmo dell'apprendimento proposto da Rosenblatt era un algoritmo iterativo:

Ad ogni iterazione t , al perceptrone viene presentato un vettore di input x_t ; il perceptrone ne calcola l'output con una funzione $f(x_t)$ e lo confronta con il risultato $g(x_t)$, andando poi ad aggiornare il vettore dei pesi w_t associato al neurone attraverso la formula:

$$w_{t+1} = w_t + \alpha * g(x_t) * x_t \quad (3.1)$$

Dove α è una costante di apprendimento strettamente positiva che regola la velocità di apprendimento, detta anche *learning rate*.

All'iterazione successiva $t+1$, il nuovo input x_{t+1} verrà questa volta pesato secondo i nuovi pesi w_{t+1} . Il perceptrone ideato da Rosenblatt tuttavia aveva dei limiti: non era infatti in grado di risolvere molte tipologie di problemi, in particolare tutti quelli dove le soluzioni e i relativi dati forniti in input (il cosiddetto *training set*) non sono separabili linearmente, ovvero, in termini geometrici, problemi per cui non esiste un iperpiano che permette di separare nello spazio vettoriale degli input, quelli che forniscono un output positivo da quelli che forniscono un output negativo. Questo limite, ad esempio, rendeva questa prima rete neurale basata sul perceptrone incapace di calcolare la funzione logica or esclusivo (XOR).

Un nuovo algoritmo di apprendimento che permise alle reti neurali di continuare a sviluppare e superare questo limite fu quello di retropropagazione dell'errore o *error backpropagation* (BP) proposto nel 1986 da David E. Rumelhart, G. Hilton e R.J. Williams. Si differenzia dall'algoritmo di apprendimento proposto da Rosenblatt nell'esserne una sua generalizzazione e permettendo quindi di risolvere problemi anche se le loro soluzioni non sono linearmente separabili.

La rete neurale è quindi costituita da diversi strati o layer; il primo è il cosiddetto strato di input, interconnesso ad uno o più strati nascosti ("hidden layer"), che si occupano dell'elaborazione vera e propria, collegati a loro volta infine ad uno strato

di output che raccoglie i risultati.

L'algoritmo della BP si basa sul metodo della discesa del gradiente, che permette di determinare i punti di massimo e minimo di una funzione di più variabili. L'idea alla base dell'algoritmo della BP è di andare a modificare i pesi delle connessioni in modo da andare a minimizzare una determinata funzione E , detta funzione di costo o funzione di errore (loss function).

La funzione di costo E si può scrivere come:

$$E(w) = \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^L (y_j^{(i)} - \hat{y}_j^{(i)})^2 \quad (3.2)$$

Questa funzione come si può vedere dipende dal vettore j -esimo di output $y_j^{(i)}$ restituito come risultato dalla rete nell'iterazione i -esima (detta epoca) e dal vettore j -esimo di output desiderati $\hat{y}_j^{(i)}$ che andiamo a fornire alla rete come parte del training set. L'algoritmo si può dividere in due passi fondamentali: il primo, il forward pass, propaga l'input dato alla rete attraverso i livelli successivi, muovendosi appunto in avanti, ossia da sinistra a destra. Arrivati alla fine si confronta l'output ottenuto con quello desiderato e si calcola $E(w)$, l'errore commesso. Il secondo passo dell'algoritmo è il backward pass, dove l'errore compiuto dalla rete viene propagato all'indietro e i pesi aggiornati di conseguenza secondo la regola:

$$w_j = w_j + \Delta w_j \quad (3.3)$$

Per stabilire il valore di Δw_j attraverso il quale si aggiorna il peso, si utilizza la seguente formula che lo ricava dall'errore commesso:

$$\Delta w_j = -\alpha \frac{\partial E}{\partial w} \quad (3.4)$$

I pesi vengono inizializzati a dei valori casuali, molto piccoli rispetto ai valori che dovranno assumere in futuro ad ogni aggiornamento e generalmente compresi tra 0 ed 1. Ogni neurone degli strati successivi a quelli di input sommera' quindi i segnali provenienti da tutti i nodi dello strato precedente a lui collegati, ognuno moltiplicato per il corrispettivo peso, e con l'aggiunta di un bias. Il bias, che letteralmente vuol dire "pregiudizio", è un valore associato al singolo neurone di una rete, allo

stesso modo dei pesi, e allo stesso modo il bias viene modificato durante la fase di addestramento della rete. Compito del bias è quindi rendere variabile il valore di soglia oltre il quale il neurone si attiva venendo sommato alla somma pesata di tutti i neuroni dello strato precedente prima di venire passata alla funzione di attivazione. La funzione di attivazione è proprio ciò che determina se un neurone andrà ad attivarsi o meno: imita, quindi, ciò che corrisponde alla soglia di attivazione nel neurone biologico. Inoltre, la funzione di attivazione è ciò che permette alle reti neurali artificiali di approssimare qualsiasi tipo di funzione esistente introducendo un fattore di non linearità, e quindi di risolvere una ampia classe di problemi come già illustrato in precedenza. Senza una funzione di attivazione infatti, limitandosi a propagare attraverso gli strati di neuroni semplicemente la somma pesata dei segnali in ingresso, una rete neurale artificiale equivarrebbe ad un modello di regressione, ovvero cercherebbe semplicemente di approssimare la distribuzione dei dati in input ad una retta.

Si possono utilizzare diverse tipologie di funzioni di attivazione per diversi strati della rete, le più comunemente utilizzati sono principalmente due: la funzione sigmoide (o softmax a seconda del tipo di problema di classificazione affrontato) e la funzione ReLu.

La funzione sigmoide è una variante della funzione a gradino, che è quella che più si avvicina ad imitare il funzionamento del neurone biologico. Per tutti i valori negativi la funzione a gradino restituirà in output il valore 0, per tutti quelli positivi il valore 1. La funzione a gradino è semplice da calcolare ed inoltre normalizza i valori in output tra i valori di 0 ed 1. La funzione a gradino, tuttavia, non è differenziabile nel punto in cui cambia valore, requisito essenziale per l'ottimizzazione della rete tramite il metodo della discesa del gradiente illustrato in precedenza. La funzione a gradino è perciò praticamente inutilizzata, in quanto rende imprevedibile il comportamento della rete. Al suo posto si utilizza principalmente la funzione sigmoide menzionata in precedenza e definita in Figura 3.1: simile a quella a gradino, il passaggio tra i valori di 0 ed 1 assume però un andamento più graduale e non presenta punti di discontinuità, evitando i problemi della funzione a gradino. Anche la funzione sigmoide ha, tuttavia, i suoi svantaggi: è caratterizzata da una convergenza molto lenta, visto che per valori molto grandi la curva ha un andamento quasi piatto, e

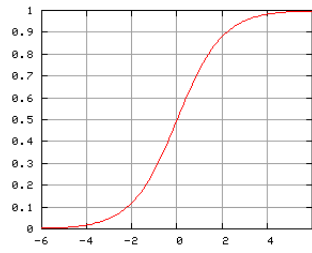


Figura 3.1: Grafico della funzione sigmoide

tende quindi a causare il cosiddetto problema della scomparsa del gradiente. Non è quindi più molto utilizzata per gli strati intermedi della rete, ma è ancora utilizzata come funzione di attivazione dello strato di output, soprattutto per problemi di classificazione binaria.

Altra funzione il cui uso è ormai diventato estremamente diffuso è la funzione softmax, molto utilizzata poichè restituisce una distribuzione di probabilità per ogni nodo di output della rete ed è quindi particolarmente usata per problemi di classificazione multi-classe. Tuttavia, nei casi di classificazione binaria, la funzione softmax diventa matematicamente equivalente alla sigmoide.

La funzione ReLu (Rectified Linear Unit) è invece ampiamente utilizzata per gli strati intermedi della rete, in particolare per la facilità con cui la si può calcolare. Il grafico della funzione ReLu è visibile in 3.2. Come si può vedere, restituisce 0 per

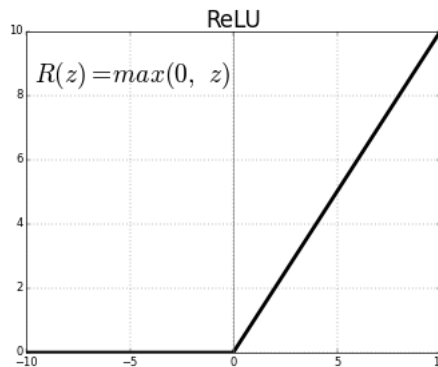


Figura 3.2: Grafico della funzione ReLu

tutti i valori negativi e lascia invariati i valori pari o superiori allo 0.

Non è inoltre soggetta agli stessi problemi della sigmoide e quindi riduce drasticamente il problema della sparizione del gradiente. Problema opposto alla scomparsa del gradiente è il problema dell'esplosione del gradiente, in cui i gradienti diventano

sempre più grandi ad ogni iterazione dell'algoritmo e i pesi subiscono aggiornamenti drastici che vanno a modificare completamente il comportamento della rete.

Una rete neurale può differenziarsi da un'altra in base a come vengono interconnessi i neuroni di uno strato con quelli dello strato successivo: quando ogni neurone di uno strato è connesso a tutti i neuroni dello strato successivo per tutti gli strati della rete, si parla di una "rete neurale fully connected" (FC). Tuttavia, per risolvere problemi relativi all'analisi e riconoscimento di immagini è necessario introdurre un'altra tipologia di rete neurale.

3.1 Reti neurali convoluzionali

L'architettura di una rete neurale convoluzionale, detta anche convolutional neural network (CNN) è ispirata all'organizzazione della corteccia visiva degli animali. In un paper di Hubel e Wiesel del 1968 venne studiato come la corteccia visuale di animali come gatti e scimmie contenesse neuroni che individualmente rispondono soltanto a piccole regioni del campo visivo. Viene definito come "campo ricettivo" di un neurone la regione del campo visivo entro cui lo stimolo riesce a influenzare l'innescamento di quello specifico neurone. Un insieme di campi ricettivi vanno a coprire nel loro insieme l'intero campo visivo, sovrapponendosi dove necessario. Il nome "rete neurale convoluzionale" viene dall'eseguire una operazione di convoluzione anziché una semplice moltiplicazione tra matrici.

Una normale rete neurale feed forward fully connected, a causa della struttura intrinseca delle sue connessioni tra neuroni, implica che il risultato di ogni neurone dipenda e sia influenzato dal valore di tutti i neuroni dello strato precedente. Questo non le rende particolarmente adatte nel venire applicate al riconoscimento di features nelle immagini per svariati motivi.

Il primo è, ovviamente, il fattore dovuto all'alto costo computazionale. Visto che ogni immagine si può rappresentare come una matrice di pixel, in una architettura FC composta anche da un unico strato nascosto (conosciuta come shallow neural network, l'opposto di una deep neural network) sarebbe richiesto un altissimo numero di neuroni e relativi pesi correlati anche per immagini di dimensioni molto piccole. Con una immagine di 100 x 100 pixel, per esempio, uno strato completamente

connesso richiederebbe l'uso di $100 \times 100 = 10'000$ pesi associati ad ogni neurone dello strato.

Inoltre questa supposizione non rispecchia la realtà: in una immagine, il valore di un pixel non è influenzato da quello di tutti gli altri pixel presenti nell'immagine, ma soltanto di quelli circostanti, per cui l'uso di strati fully connected per il riconoscimento di features in una immagine diventerebbe non solo computazionalmente oneroso ma anche non corretto.

Lo scopo di una rete convoluzionale è proprio quello di ridurre le immagini in una forma più facile da processare, cercando di ridurre la perdita di informazioni sull'immagine andando ad estrarne le features. In una rete convoluzionale a più strati, il primo strato ha generalmente il compito di estrarre features di basso livello come contorni o colori dell'immagine, ed ogni strato convoluzionale aggiuntivo va ad estrarre ed apprendere features via via più generiche, fino ad arrivare a features di alto livello che permettono il riconoscimento completo di uno specifico soggetto e una "visione" completa dell'immagine.

Una volta estratte le features dall'immagine, si possono utilizzare dei livelli fully connected che andranno ad eseguire la classificazione vera e propria.

Il funzionamento in dettaglio di uno strato convoluzionale avviene come descritto di seguito.

Immaginiamo di avere come input della rete una immagine di dimensione 5 (Larghezza) x 5 (Altezza) pixel, che può essere rappresentata come una matrice dove ogni elemento $a_{i,j}$ della matrice rappresenta il pixel in posizione i,j -esima. Una seconda matrice di dimensioni minori $n \times n$ detta kernel o filtro, viene fatta "scorrere" sopra la matrice iniziale eseguendo una operazione di moltiplicazione tra matrici tra il kernel e la porzione di immagine attualmente coperta. Il filtro partirà dall'angolo in alto a sinistra dell'immagine, che corrisponde all'elemento di indice 1,1 della matrice, e procederà a "scorrere" l'immagine per tutta la sua lunghezza, spostandosi verso destra di una quantità detta passo o stride.

Dopodiché torna all'inizio dell'immagine e scende di un numero di righe pari sempre al passo scelto, finché non ha attraversato tutta l'immagine. Al termine del passaggio del filtro si otterrà una matrice che conterrà i risultati di tutte le operazioni di moltiplicazione tra matrici effettuate. Un esempio del procedimento appena illustrato

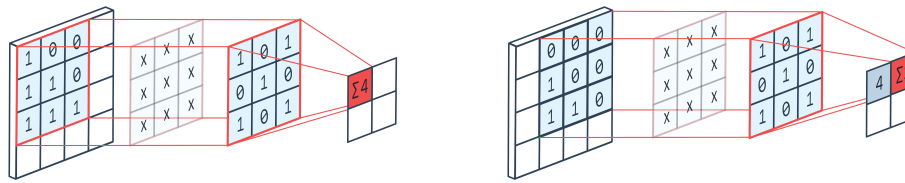


Figura 3.3: Un esempio di convoluzione con kernel 3x3 e passo=1

è visibile nella Figura 3.3. Questa matrice è detta Convolutional Feature Output e rappresenta quanto il nostro filtro è stato in grado di riconoscere una determinata feature nell'immagine. Nel caso di immagini a più canali come le immagini RGB, il kernel avrà la stessa profondità del numero dei canali dell'immagine. Ogni canale avrà il suo filtro e il risultato di ogni canale viene sommato con l'aggiunta di un bias per ottenere una Convolutional Feature Output complessiva di tutta l'immagine, che sarà di dimensioni minori dell'immagine di partenza.

In questo modo abbiamo condensato e riassunto l'informazione contenuta nell'immagine per ridurre il costo computazionale necessario al processare i dati.

Successivo allo strato di convoluzione troviamo il cosiddetto strato di Pooling. Così come lo strato di Convoluzione aveva il compito di ridurre la dimensione della matrice su cui lavorare, scopo dello strato di Pooling è quello di ridurre ulteriormente la dimensione del Convolutional Feature Output. Il pooling, inoltre, ha anche la funzione di estrarre feature che sono invariati alla posizione e alla rotazione, continuando l'addestramento del modello. Il funzionamento dello strato di pooling è simile a quello di convoluzione: un filtro scorre lungo tutto il Convolutional Feature Output ed esegue una diversa operazione a seconda del tipo di pooling effettuato.

Vi sono generalmente due tipologie di pooling possibili: l'Average Pooling e il Max Pooling.

L'Average Pooling da come risultato la media tra tutti i valori della porzione di immagine coperta dal filtro, mentre il Max Pooling restituisce il massimo tra tutti i valori presenti, ovvero quello che meglio ha "rappresentato" la feature individuata. Mentre l'Average Pooling si limita a ridurre la dimensione dell'immagine, il Max Pooling scartando completamente tutti i valori non massimi effettua anche una operazione di riduzione del rumore. Per questo motivo, è generalmente preferito il Max Pooling all'Average Pooling. La differenza tra le due tipologie di Pooling

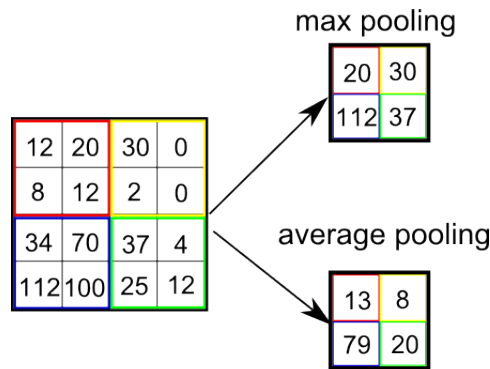


Figura 3.4: Differenza tra Average Pooling e Max Pooling

è illustrata nella Figura 3.4. Su una immagine vengono passati un numero molto elevato di filtri, ognuno atto a individuare la presenza di una specifica feature come linee orizzontali, verticali o angoli dell'immagine. Le diverse feature vengono raccolte in un unico vettore denominato come feature map.

Al termine dell'alternanza di strati convoluzionali e strati di pooling ripetuta più volte, tutti i vettori di feature ottenuti vengono appiattiti in un unico grande vettore attraverso una operazione denominata come appiattimento (flatten), e poi dati in input ad altri strati completamente connessi per l'operazione di classificazione vera e propria. In Figura 3.5 si può vedere uno schema riassuntivo di tutti gli strati illustrati fino ad ora di una CNN.

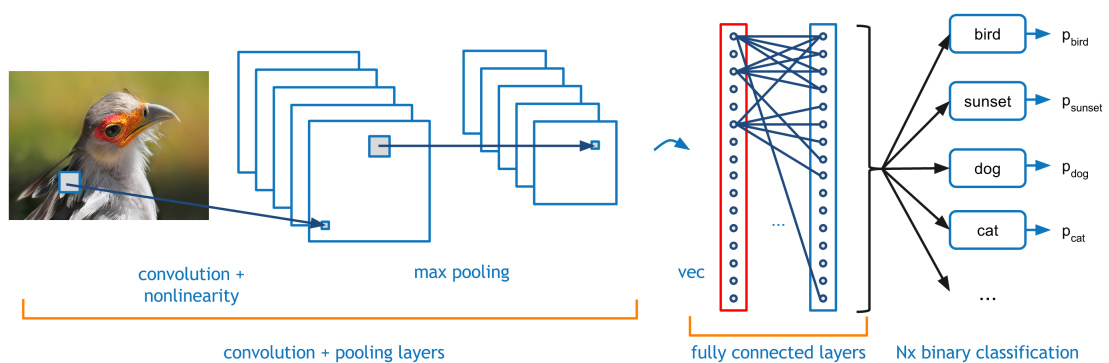


Figura 3.5: Schema di una CNN completa, dall'immagine di input all'output finale

3.1.1 Reti neurali convoluzionali 3D

Tra le varie tipologie esistenti di CNN, le reti neurali convoluzionali 3D sono di sviluppo relativamente recente e continuano a essere oggetto di studi. Le 3D

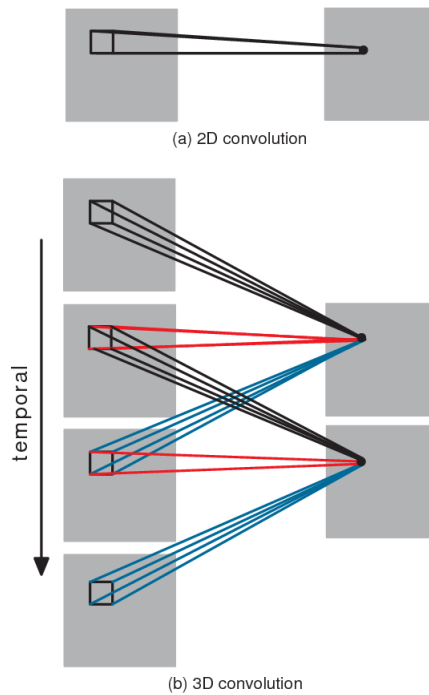


Figura 3.6: Feature maps in convoluzione 2D e 3D

Convolutional Neural Network (o 3DCNN), anzichè operare su una immagine a due dimensioni operano in tre dimensioni (altezza, larghezza, profondità); questo permette loro di operare in contesti dove una singola immagine non è sufficiente a rappresentare tutta l'informazione necessaria (come fanno le normali 2D CNN), ma occorre anche tenere conto del contesto temporale o spaziale in cui si trovano o, in altre parole, occorre esaminare gruppi di immagini nel loro insieme. Esempi di campi dove si sta studiando l'uso di 3DCNN sono l'analisi di video (rappresentati come una sequenza di frame costituiti dalle singole immagini) e l'analisi di immagini mediche. Sono strutturalmente identiche alle 2DCNN, ovvero formate da una sequenza di strati di convoluzione e strati di pooling alternati per l'estrazione di feature maps, a cui seguono strati completamente connessi per la classificazione, con la differenza dell'utilizzare una dimensione aggiuntiva negli strati di convoluzione e di pooling.

Anzichè operare su una immagine rappresentata come matrice di dimensioni $n \times n$, gli strati convoluzionali 3D fanno passare un filtro a tre dimensioni su un insieme di immagini successive tra di loro, come potrebbero esserlo i frame di un video, rappresentate da una matrice $n \times n \times n$. In questo modo, come si può vedere in 3.6 la feature maps viene generata dalle immagini contigue e va ad estrarre feature che

rappresentano l'informazione contenuta non in una singola immagine ma nel loro insieme. Funzionamento perfettamente analogo hanno anche gli strati di pooling che seguono quelli di convoluzione che hanno lo stesso compito di ridurre la dimensione dei dati su cui lavorare. In questa tesi abbiamo appunto fatto uso di una rete convoluzionale 3D, precisamente una C3D pre-addestrata a lavorare con frame di video, e l'abbiamo applicata alle TAC polmonari, le quali sono costituite da una serie di immagini bidimensionali, dette slices, assimilabili ad un video breve.

3.1.2 Reti neurali ConvLSTM

Un'altra tipologia di reti neurali sono le reti neurali ricorrenti o ricorsive (Recurrent Neural Network, RNN). L'idea che sta alla base di una RNN [16] è una rete neurale feedforward che ha al suo interno un meccanismo che le permette di avere una "memoria" dell'iterazione precedente invece di aggiornare semplicemente pesi e bias dei neuroni della rete. Prodotto un output al termine di una iterazione, l'output viene "re-inviato" (da qui ricorsive) all'interno della rete e prende il nome di *hidden state*, e l'output dell'iterazione successiva andrà a dipendere dagli input in ingresso e dall'hidden state che ha ottenuto dai dati precedenti. Le RNN sono quindi un altro modo per elaborare sequenze di input andando ad esaminarle nel loro insieme, in quanto gli input diventano collegati e dipendenti l'uno dall'altro. Uno dei problemi tuttavia delle RNN è costituito dalla difficoltà nell'addestrarle e dall'essere suscettibili al problema dell'esplosione del gradiente. Sono inoltre soggette al problema dell'essere fornite unicamente di una "memoria a breve termine", ovvero se una sequenza data in input alla rete è particolarmente lunga possono avere difficoltà a "ricordare" l'informazione dei primi elementi della sequenza per tutta la sua lunghezza. Immaginiamo ad esempio di voler addestrare una RNN ad effettuare una analisi del testo per predire la parola successiva in un determinato punto della frase. Per svolgere questo compito la rete potrebbe avere bisogno in un determinato caso di informazioni utili a capire il contesto provenienti dall'inizio della frase, che però avrà "dimenticato" finché ha raggiunto il termine della sequenza. Questo problema ed altri vengono risolti da una variante delle RNN, le Long Short Term Memory network (LSTM). Come indicato dal nome, le LSTM si basano sul facilitare il "ricordo" dell'informazione ottenuta precedentemente dalla rete attraverso dei meccanismi interni noti come

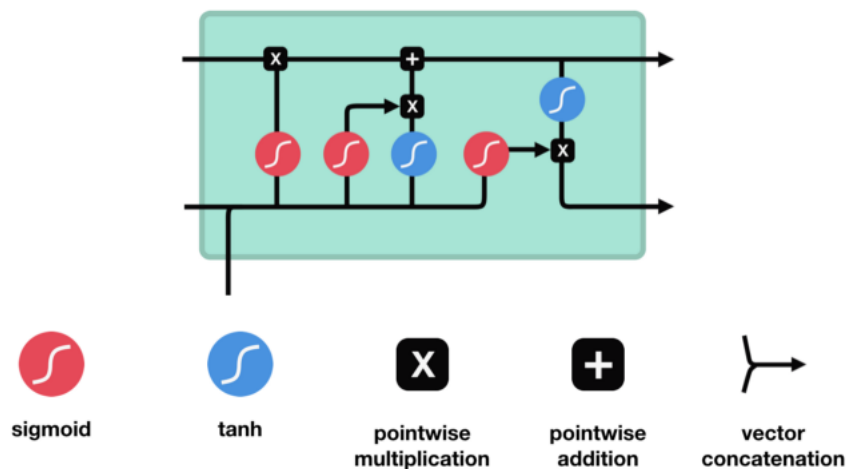


Figura 3.7: Struttura di una cella LSTM

"cancelli" (gates) che permettono alla rete di imparare quali dati in una sequenza è importante ricordare e quali invece si possono "dimenticare". Per capire meglio il funzionamento di una rete LSTM possiamo fare riferimento alla Figure 3.7. Il concetto alla base di una rete LSTM è lo stato interno della sua cella; la cella si può immaginare come la "memoria" della rete. Attraversando la sequenza di dati in input, nuove informazioni vengono aggiunte ("ricordate") oppure rimosse ("dimenticate") attraverso i vari cancelli interni alla cella, che in questo modo può "trasportare" dati utili provenienti dai primi elementi di una sequenza fino alla fine.

Il primo "cancello" attraverso il quale passano le informazioni è il cosiddetto *forget gate*; tutti i cancelli utilizzano una sigmoide come funzione di attivazione. In questo modo, i valori in ingresso vengono ridotti ad un valore compreso tra 0 ed 1. Nel caso del *forget gate*, più vicino a 0 è il valore, meno importante da ricordare sarà l'informazione. Un valore vicino o pari ad 1, invece, indica che l'informazione è estremamente importante e va ricordata.

Altro cancello è l'*input gate*, che decide come andrà aggiornato lo stato interno della cella. Innanzitutto si passa l'input attuale e l'hidden state precedente attraverso una funzione sigmoide. Anche qui, un risultato tanto più vicino ad 1 indica che l'informazione in ingresso è importante e lo stato della cella va aggiornato.

Infine abbiamo l'*output gate*, che sempre attraverso l'utilizzo di una funzione sigmoide unita ad un'altra funzione di attivazione tanh andrà a decidere quale informazione

costituirà il prossimo hidden state da utilizzare nell'elemento successivo della sequenza. Riassunto: il forget gate decide cosa è importante ricordare dagli elementi precedenti della sequenza, l'input gate quale informazione è importante ricordare dall'elemento attuale della sequenza e l'output gate "condensa" il tutto andando il prossimo hidden state. Una rete ConvLSTM combina la cella di memoria di una rete LSTM con le operazioni di convoluzione delle CNN (invece di semplici moltiplicazioni tra matrici di una normale RNN), permettendo in questo modo di estrarre e ricordare informazione attraverso gruppi di immagini appartenenti ad una sequenza, come potrebbero esserlo appunto frame di un video o una sequenza di immagini mediche.

Capitolo 4

Valutazione dell'apprendimento

Dopo aver illustrato il funzionamento teorico di una rete neurale e come opera, è necessario passare ad illustrare le modalità con cui è possibile interpretare i risultati ottenuti dalla rete e valutarne la loro qualità su base oggettiva, in altre parole, quantificare il grado di apprendimento che la rete ha raggiunto in seguito alla fase di addestramento.

Per far ciò è necessario introdurre alcuni concetti di analisi e classificazione statistica. La prima modalità a cui si potrebbe pensare per valutare l'apprendimento effettuato dalla rete sarebbe quello di confrontare i risultati predetti su un campione di dati denominato di test da quelli corretti e valutare la percentuale di predizioni corrette. Questo è il caso dell'apprendimento supervisionato menzionato in precedenza ma non è l'unico indice importante di cui tener conto.

4.1 La matrice di confusione

Nell'ambito dell'intelligenza artificiale la matrice di confusione, detta anche matrice di errata classificazione, è una tabella che permette di interpretare l'accuratezza di una classificazione statistica. E' applicabile a casi di classificazione multi classe, ma per semplicità e diretta correlazione al nostro lavoro, illustreremo un esempio di classificazione binaria.

La matrice di confusione si può rappresentare come una matrice di dimensione $n \times n$, dove n è il numero di classi che il nostro classificatore deve essere in grado di distinguere. In questa tesi, dove il nostro obiettivo è classificare tra tumore adenocarcinoma e tumore a cellule squamose, si trarrà quindi di una matrice 2×2 .

	Predicted	
	Positive	Negative
Actual True	TP	FN
Actual False	FP	TN

Figura 4.1: Un esempio di matrice di confusione di classificazione binaria

Ogni riga della matrice rappresenta i valori reali, mentre le colonne i valori predetti. Tuttavia, nell'ambito dell'apprendimento automatico e' più conveniente ed immediato usare una rappresentazione alternativa basata su quattro casi possibili di classificazione rappresentati in 4.1. In una classificazione binaria per definizione le etichette che identificano le classi possono assumere solo due valori. Supponiamo si identifichi la prima classe con il valore '1' o positivo e la seconda classe con il valore '0' o negativo. Nella matrice di confusione sono rappresentati quattro casi possibili:

- **TP**: True Positive o veri positivi, ovvero un campione della classe 1 viene classificato correttamente come tale.
- **FP**: False Positive o falsi positivi, ovvero un campione della classe 0 viene erroneamente classificato come campione di classe 1.
- **FN**: False Negative o falso negativo, un campione della classe 1 viene erroneamente classificato come campione di classe 0.
- **TN**: True Negative o vero negativo, un campione della classe 0 viene classificato correttamente come tale.

Ogni predizione effettuata dal classificatore rientrerà in una di queste quattro tipologie: ovviamente, per ottenere un classificatore accurato si cercherà di ridurre al minimo il numero di falsi positivi e falsi negativi che si verificano.

Il falso positivo è detto anche errore di tipo 1 mentre il falso negativo errore di tipo 2. A seconda del classificatore, generalmente l'errore di tipo 2 è più grave dell'errore di tipo 1. Immaginando ad esempio di avere un classificatore che distingue tra tumore maligno (classe 1) e benigno (classe 0), è molto più grave e dannoso per il paziente che si verifichi un falso negativo, ovvero un caso di tumore maligno non riconosciuto

e classificato come benigno, rispetto al contrario, ovvero un caso di tumore benigno erroneamente classificato come maligno.

Nel primo caso il paziente viene falsamente rassicurato e il tumore può passare inosservato fino a raggiungere stadi in cui non è più trattabile, nel secondo caso invece i test aggiuntivi richiesti andranno invece a confermare che il tumore non è maligno. Calcolando il numero di TP,FP,FN e TN che si verificano nel classificatore è possibile andare a calcolare le metriche utilizzate per valutarne le prestazioni.

4.2 Accuratezza

La prima metrica che si va a valutare è l'accuratezza, definita come

$$ACC = \frac{TP + TN}{TP + FP + FN + TN} \quad (4.1)$$

ovvero il numero di veri positivi e veri negativi che costituisce tutte le classificazioni corrette diviso il numero totale di classificazioni. La accuratezza non è tuttavia sufficiente a rappresentare correttamente le prestazioni di un classificatore nel caso di sbilanciamento tra le due classi nel dataset, o se siamo più interessati ad una classe rispetto ad un'altra. Riprendendo l'esempio precedente, se il nostro dataset è fortemente sbilanciato presentando un gran numero di casi di tumore benigno e pochissimi di tumore maligno, il classificatore risulterà comunque nel complesso estremamente accurato perché semplicemente imparerà a predire la classe "tumore benigno" in tutti i casi, non riuscendo invece a classificare correttamente la classe "tumore maligno".

4.3 Precisione e recupero

Per ovviare a questa mancanza dell'accuratezza vengono introdotte altre due metriche, la prima delle quali è la precisione o *precision* così definita:

$$Precision = \frac{TP}{TP + FP} \quad (4.2)$$

ovvero il numero di veri positivi classificati correttamente diviso il numero totale di classificazioni positive, anche quelle non corrette. Per questo motivo è nota anche

come *positive predicted value*.

Usato spesso insieme alla precisione è anche il valore di recupero o *recall* definito come

$$Recall = \frac{TP}{TP + FN} \quad (4.3)$$

Il richiamo si può vedere come il tasso di veri positivi o *true positive rate* del classificatore, in quanto è il rapporto tra i veri positivi classificati correttamente e il numero totale di classificazioni che avrebbero dovuto essere classificati come positivi. Chiaramente, un classificatore con precision e recall entrambi pari ad 1 sarebbe un classificatore senza falsi positivi o negativi, ovvero accurato al 100%.

4.4 F1 score

Spesso calcolato andando a combinare i valori di precision e recall è l'F1 score, definito come la media armonica dei due valori. Il suo valore è chiaramente pari ad 1 se sia precision che recall sono pari ad 1, e 0 se uno dei due è uguale a 0, e definito in (4.4)

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (4.4)$$

che si può riscrivere andando a sostituire le rispettive definizioni come

$$F1 = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \quad (4.5)$$

L'F1 score è utile per fornire una valutazione complessiva che tiene conto sia della precision che della recall; limitarsi ad osservare una sola delle due metriche potrebbe infatti portare a conclusioni errate sul comportamento assunto dalla rete.

L'F1 score viene utilizzato anche quando si devono andare a confrontare due diversi classificatori, il primo ad esempio con buona recall ma scarsa precision e il secondo con valutazione opposta; comparare univamente precision e recall non permetterebbe di capire quale classificatore sia effettivamente migliore, perciò viene utilizzato l'F1 score come valore univoco.

Uno possibile svantaggio collegato all'uso del F1 score come metrica di valutazione per classificatori binari è che, come è possibile vedere dalla formula, non tiene però conto

dei veri negativi, che rende questa metrica più utile in compiti come l'information retrieval dove il numero di veri negativi non è calcolabile.

4.5 ROC e AUC

Altra metrica utile per valutare la performance di un classificatore è il grafico della curva ROC (*receiver operating characteristic*). Come illustrato in [17], i grafi ROC sono utili per visualizzare la performance di un classificatore in modo da poterla valutare e confrontare con altri classificatori. Prima di tutto è necessario introdurre altri due valori utilizzati nei grafi.

Il primo è la *true positive rate*, che come illustrato in precedenza (4.3) coincide con la *recall* e non viene quindi nuovamente illustrato. Il secondo valore utilizzato è la *false positive rate*, detto anche *false alarm rate* di un classificatore e definito come:

$$FPR = \frac{FP}{FP + TN} \quad (4.6)$$

Il FPR corrisponde alla percentuale di falsi positivi diviso il numero totale di campioni appartenenti alla classe negativa, dato dalla somma di falsi positivi e veri negativi.

Il grafo ROC si ottiene andando a mappare questi due valori su un piano cartesiano, con TPR sull'asse delle ordinate y e FPR sull'asse delle ascisse x.

E' possibile identificare dei punti nel piano che corrispondono alla performance di un classificatore andando ad individuare i punti dei rispettivi valori di TPR e FPR. Un esempio di un grafo ROC con punti corrispondenti a cinque classificatori discreti è visibile in Figura 4.2.

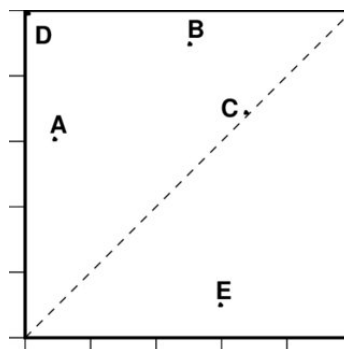


Figura 4.2: Grafo ROC

Capitolo 4 Valutazione dell'apprendimento

Il punto $(0,1)$ dove si trova il classificatore D nel nostro esempio corrisponde ad un classificatore perfetto, con una percentuale di TPR del 100% e di FPR (ovvero errori commessi nel classificare campioni appartenenti alla classe negativa) del 0%. Il punto $(0,0)$ nell'origine invece corrisponderebbe ad un classificatore che non dà mai una risposta corretta: non commette mai errori di falsi positivi ma non ottiene nemmeno veri positivi. Viceversa, il punto opposto in posizione $(1,1)$ nell'estremo in alto a destra del grafo corrisponde alla sua strategia opposta, di dare come risposta sempre e unicamente la classe positiva, risultando in un TPR del 100% ma anche in un FPR del 100%. In termini generali, un classificatore posizionato nello spazio ROC è tanto migliore quanto più a nord-ovest si trova sul grafo rispetto ad un altro. I classificatori che appaiono sul lato sinistro del grafo nei pressi dell'asse X si possono ritenere "cauti": eseguono classificazioni positive solo con una buona certezza quindi commettono pochi errori di falsi positivi, ma tendono ad avere anche un TPR basso. Al contrario i classificatori che si posizionano nella metà destra e in alto del grafo tendono a classificare molti più elementi nella classe positiva (fino all'estremo illustrato nel punto $(1,1)$ dove tutte le risposte date sono positive) con meno sicurezza; in questo modo ottengono un risultato corretto per quasi tutti i campioni effettivamente positivi, ma anche molti più falsi positivi. I classificatori che si posizionano lungo la diagonale $x=y$ rappresentano i classificatori randomici, ovvero che danno risposte casuali. Ad esempio, con un classificatore che dà come risposta la classe positiva metà delle volte e quella negativa nell'altra metà, ci si può aspettare che individui correttamente metà dei campioni positivi e metà di quelli negativi, risultando nel punto $(0.5,0.5)$ sul grafo. Un classificatore che dà come risposta la classe positiva nel 90% dei casi ci si aspetta individuerà correttamente circa 90% dei campioni positivi, ma avrà anche un FPR del 90%, risultando nel punto $(0.9,0.9)$ sul grafo che si trova sempre sulla diagonale $x=y$. Possiamo vedere come i due $(0,0)$ e $(1,1)$ non sono altro che le due estremizzazioni possibili di un classificatore casuale. Infine, i classificatori che si posizionano nell'estremo sud-est del grafo, al di sotto della diagonale, possono venire descritti come classificatori che hanno un comportamento peggiore del dare risposte casuali, e spesso questa sezione del grafo ROC tende a restare vuota.

Si può quindi vedere come semplicemente osservando la posizione di un classificatore

sul piano ROC è possibile ottenere informazioni utili sul comportamento del classificatore; un classificatore che si posiziona sulla diagonale, quindi casuale, si può dire non essere stato in grado di ottenere informazioni dai dati che gli permettono di distinguere correttamente le due classi, ovvero non è stato in grado di apprendere dai dati. Un classificatore al di sopra della diagonale ha appreso tanto meglio dai dati quanto più vicino all'estremo (0,1) si trova. Un classificatore con comportamento peggiore del casuale ha appreso dai dati, ma le sta appositamente applicando nel modo sbagliato.

4.5.1 Curve ROC

Abbiamo illustrato come un classificatore discreto che restituisce una singola valutazione per ogni elemento del test, ottenie quindi un singolo punto nello spazio ROC. Classificatori non discreti come le reti neurali, tuttavia, restituiscono la probabilità di una istanza, ovvero per ogni elemento del test set la probabilità di quell'elemento di appartenere alla classe positiva o negativa e quindi la certezza della risposta. Ad esempio, per un campione appartenente alla classe positiva potrebbe restituire come risposta il 70% di probabilità che appartiene alla classe positiva e 30% alla classe negativa. Un classificatore di questo tipo si può comunque rappresentare nello spazio ROC utilizzando un *threshold*, ovvero un valore di soglia, per ricondurlo ad un classificatore discreto: se il valore restituito dal classificatore supera la soglia, equivale al classificare un campione come positivo, altrimenti come negativo. Variando il valore di soglia si ottengono punti diversi sullo spazio ROC, che vanno a formare una curva come mostrato in 4.3.

La curva (in realtà una funzione a gradino, che si va ad approssimare ad una curva quanti piu' campioni si hanno a disposizione) ROC così generata va quindi a rappresentare la capacità del classificatore di discernere correttamente i campioni positivi rispetto a quelli negativi al variare dei valori di soglia.

4.5.2 AUC

Una curva ROC è una rappresentazione della performance di un classificatore in uno spazio bidimensionale. Tuttavia, per confrontare classificatori diversi puo' essere piu' conveniente poter utilizzare un singolo valore scalare. Per fare ciò si calcola

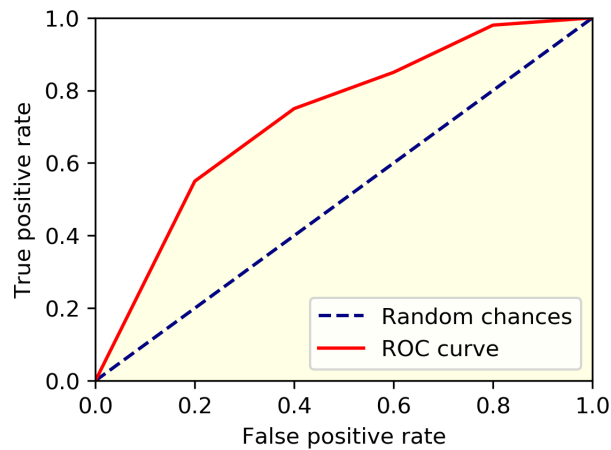


Figura 4.3: Un esempio di curva ROC

spesso l'area al di sotto della curva ROC, o *Area Under the Curve* (AUC). Siccome l'AUC sarà sempre una porzione del grafo che è equivalente ad un quadrato di lato unitario, anche il suo valore sarà sempre compreso tra 0 e 1.0. Ricordando sempre che la diagonale $x=y$ rappresenta un classificatore casuale, generalmente un buon classificatore non dovrebbe avere AUC inferiore a 0.5, in quanto corrisponderebbe all'area del grafo dove un classificatore restituisce risposte appositamente sbagliate, un comportamento ovviamente non desiderato. Come illustrato in [18], il valore AUC indica la capacità del classificatore di discernere correttamente le classi riassumendola in un valore unico.

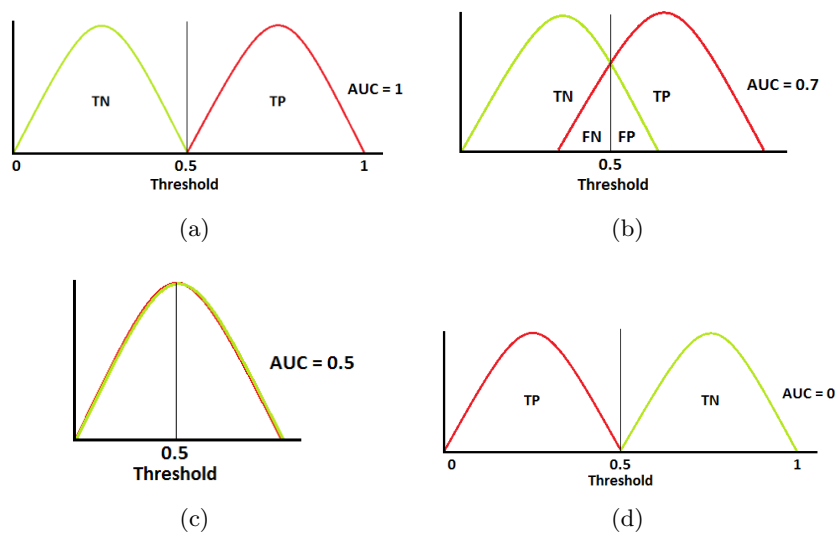


Figura 4.4: Un esempio di quattro valori AUC

Possiamo rappresentare su un grafo attraverso due curve la distribuzione di veri positivi e veri negativi (ovvero le risposte corrette date) in relazione al valore di soglia e il corrispondente AUC, come illustrato in 4.4.

Un $AUC = 1$ corrisponde al caso migliore: le due classi sono completamente separate e il classificatore riesce a distinguerle chiaramente in tutti i casi. In 4.4(b) possiamo vedere che ad un $AUC = 0.7$ inizia ad esserci una sovrapposizione tra le due classi: iniziano quindi ad esserci campioni classificati come falsi positivi e falsi negativi, che possono variare a seconda della soglia. Un valore AUC di 0.7 indica che il classificatore riesce a distinguere tra le due classi nel 70% dei casi. 4.4(c) e 4.4(d) indicano, rispettivamente, i due casi peggiori illustrati in precedenza. Con $AUC = 0.5$ le due classi sono completamente sovrapposte, ovvero il classificatore non riesce a distinguerle. Un $AUC = 0$ corrisponde all'aver completamente invertito le due classi, ovvero classificare sempre un campione positivo come negativo e viceversa.

Capitolo 5

Tomografia assiale computerizzata

Nel Capitolo 1 abbiamo accennato al dover far uso di TC nell'analisi delle immagini per tentare di individuare la tipologia di cancro al polmone. Introduciamo quindi più in dettaglio cosa sono le TC, lo standard DICOM utilizzato per rappresentarle ed il processo di preparazione delle immagini necessario, detto anche *preprocessing* delle immagini.

5.1 Tomografia Computerizzata

La TC o Tomografia Computerizzata è una tecnica di diagnostica attraverso le immagini che permette di esaminare qualsiasi parte del corpo allo scopo di individuare tumori ed altre patologie. Il termine "tomografia" indica una tecnica spettroscopica che mira alla creazione di una rappresentazione a strati del corpo umano, in contrasto alla radiografia convenzionale che rappresenta su una lastra bidimensionale tutta lo spessore dell'oggetto o corpo umano, e si basa sui principi della geometria delle proiezioni: da proiezioni di un oggetto da molte direzioni differenti (idealmente infinite per massima precisione) è possibile ricostruire l'oggetto esaminato. Rielaborando al computer queste tomografie (da lì tomografia computerizzata), si ottiene una immagine tridimensionale dell'oggetto. Le immagini sono generate attraverso l'attenuazione di un fascio di raggi X mentre attraversa una sezione o l'intero corpo, basandosi sulla legge dell'assorbimento dei raggi X [19]: dato un fascio di raggi X di intensità iniziale I_0 , esso viene attenuato di una quantità $I(t)$ in una misura che decresce esponenzialmente rispetto al coefficiente di attenuazione di massa μ e al cammino percorso nel mezzo t . Il coefficiente μ dipende a sua volta

dalla densità ρ del materiale e dalla energia E del fascio di raggi X. La legge si può riassumere nella formula:

$$I(t) = I_0 e^{-\mu t} \quad (5.1)$$

Un fascio di raggi X che attraversa quindi l'oggetto verrà attenuato tanto più quanto attraversa materiali con un alto numero atomico; maggiore l'attenuazione più bassa sarà l'energia e maggiore quindi lo spessore attraversato. Viceversa, attraversando un materiale a densità minore, minore è lo spessore attraversato, minore è anche l'attenuazione e più alta sarà invece l'energia del fascio. Per questo motivo nelle immagini TC questo porta gli oggetti con densità maggiore e quindi massima attenuazione (ad esempio le ossa) ad apparire chiari, mentre gli oggetti con densità minore (come i polmoni, che contengono molta aria) appariranno scuri poichè hanno attenuazione minima. Un esempio di una immagine TC si può vedere in 5.1. A volte, per ottenere immagini migliori della vascolarizzazione di organi e

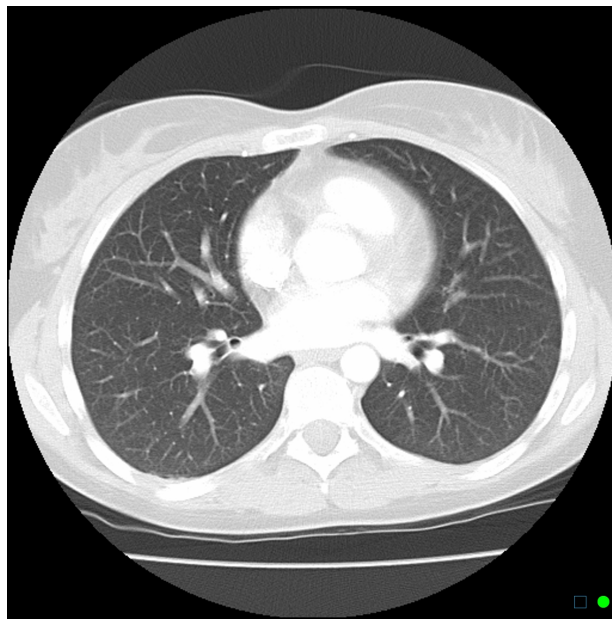


Figura 5.1: Esempio di una TC assiale toracica

tessuti, prima dell'esame viene iniettato per via endovenosa un mezzo di contrasto a base di iodio.

5.2 Scala di Hounsfield

Per confrontare i risultati ottenuti da strumentazioni diverse si utilizza la scala di Hounsfield [20], usata per descrivere quantitativamente la radiodensità. In ogni *voxel* (punto a tre dimensioni) dell'immagine si può misurare il *numero CT*, misurato in Hounsfield Units (HU) e che rappresenta la densità dell'oggetto in quel punto attraverso la formula (5.2)

$$CT\ number = 1000 * \frac{\mu - \mu_{H_2O}}{\mu_{H_2O}} \quad (5.2)$$

Si prende come riferimento il coefficiente di attenuazione di massa dell'acqua, che avrà quindi $CT\ number = 0$ HU. La densità dell'aria è considerata nulla ($\mu = 0$) e avrà quindi un $CT\ number = -1000$ HU. Un materiale come l'osso, con una densità circa doppia a quella dell'acqua, avrà un $CT\ number = +1000$ HU e così via. L'acqua come materiale di riferimento è stata scelta per via della sua presenza in grandi quantità nel corpo umano e di altri esseri viventi. Valori in HU di altri tessuti presenti nel corpo umano sono illustrati in Tabella 5.1

Sostanza	HU
Aria	-1000
Polmone	-500
Tessuto adiposo	da -100 a -50
Acqua	0
Fluido cerebrospinale	15
Rene	30
Sangue	da +30 a +45
Tessuto muscolare	da +10 a +40
Materia grigia	da +37 a +45
Materia bianca	da +20 a +30
Fegato	da +40 a +60
Mezzo di contrasto	da +100 a +300
Osso	da +400 a +3000

Tabella 5.1: Valori HU di alcune sostanze

Capitolo 6

Standard DICOM e preprocessing dell'immagine

Lo standard DICOM [21] (Digital Imaging and Communications in Medicine) definisce criteri per visualizzare, trasmettere, archiviare e stampare informazioni biomediche. E' ormai impiegato nella quasi totalità dei dispositivi utilizzati in ambito medico: definisce sia i protocolli da utilizzare nella trasmissione delle informazioni sia il formato utilizzato per l'archiviazione di immagini mediche digitali e relativi dati correlati. Nello standard DICOM qualsiasi oggetto del mondo reale come potrebbe esserlo un ricovero, una immagine od un paziente vengono definiti come un oggetto, ispirando al paradigma Object Oriented (OO) della programmazione informatica. Ogni oggetto conterrà una serie di attributi; definiti ogni tipologia di oggetto e relativi attributi, lo standard DICOM definisce per ogni oggetto quali operazioni è possibile eseguirvi. In questo modo si assicura che lo stesso oggetto operi nello stesso modo su qualsiasi dispositivo che rispetti lo standard DICOM, permettendo operazioni cross-platform. Tra i vari oggetti possibili, i dati che vengono rappresentati come immagini e archiviati secondo lo standard DICOM vengono comunemente chiamate "immagini DICOM". Come accennato ogni oggetto nello standard DICOM presenta degli attributi; nel caso di una immagine DICOM questi attributi sono contenuti nella intestazione (*header*) del file, e possono includere dati come nome ed ID del paziente, data di acquisizione, tipo di scansione, posizione e dimensione dell'immagine e moltissimi altri. Tutte le possibili informazioni incluse nell'intestazione sono suddivise in gruppi che prendono il nome di "Tag DICOM". Si può accedere alle informazioni contenute nell'intestazione dell'immagine attraverso appositi programmi o librerie in

grado di elaborare lo standard DICOM, manipolandole poi come se fossero qualsiasi altro tipo di dato.

6.1 Preprocessing dell'immagine

Prima di poter essere utilizzate come input per reti neurali artificiali e a scopo di diagnostica, è necessario eseguire svariate operazioni di preparazione sulle immagini che prendono il nome di "preprocessing" dell'immagine. La necessità di eseguire del preprocessing ha svariate cause: la prima e la più banale è una semplice questione di costo computazionale. Una singola TC può risultare in un insieme di centinaia di immagini ad alta risoluzione e di grandi dimensioni; fino a poco tempo fa le immagini generate erano di 512 x 512 pixel, ma ormai i dispositivi moderni riescono a generare immagini di dimensioni ancora maggiori, dell'ordine delle migliaia di pixel per singola immagine. Questa maggiore accuratezza va ovviamente a discapito delle dimensioni dell'immagine e dello spazio e potenza di calcolo necessari ad elaborarle. La seconda motivazione è che una TC completa contiene tantissima informazione che alla rete non torna utile nello scopo per cui deve essere addestrata, che quindi va a costituire unicamente "rumore" che può anzi andare a peggiorare le prestazioni della rete impedendone un corretto addestramento. Per un esempio immediato di cosa si intende per "rumore" andiamo ad osservare un'altra immagine di una TC in 6.1. Immaginando che la nostra area di interesse siano sempre i polmoni, possiamo vedere immediatamente che la nostra immagine contiene molti altri elementi che non ci interessano: prima di tutti tutta la zona "vuota" tutt'attorno che risulta completamente nera, e che contiene addirittura la sagoma del lettino su cui viene adagiato il paziente durante la tomografia. Per un medico è chiaramente ovvio ignorare le parti dell'immagine che non interessano, ma per una rete artificiale ciò non è altrettanto immediato: nel caso peggiore, una rete potrebbe addirittura addestrarsi accidentalmente a classificare immagini attraverso criteri completamente sbagliati. Se ad esempio per puro caso le immagini di una certa tipologia di tumore includono più spesso il lettino ed altre no, la rete potrebbe addestrarsi a classificare una immagine come appartenente ad una determinata tipologia di tumore semplicemente rivelando o meno la presenza della *features* "lettino", criterio ovviamente errato.

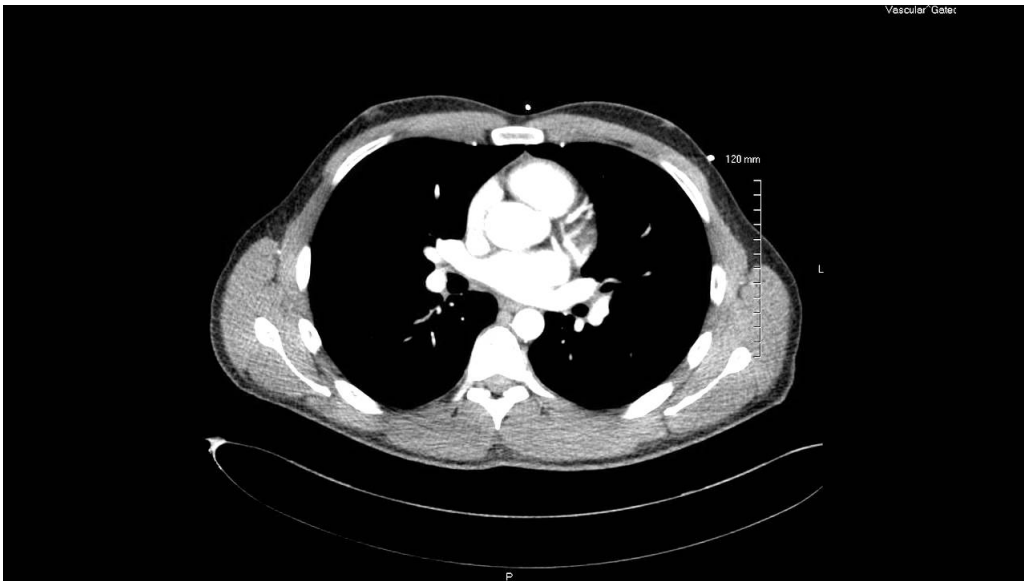


Figura 6.1: Un altro esempio di TC assiale toracica

Una prima pulizia che si può pensare di eseguire sull'immagine è di eliminare tutto ciò che non ci interessa ritagliando l'immagine. Prima di poterlo fare, tuttavia, sono necessarie alcune operazioni preliminari. Innanzitutto è necessario operare una trasformazione lineare su ogni pixel dell'immagine in modo da convertirne il valore nella scala HU, così da poter usare i valori della scala HU specificati in precedenza per sapere che tipologia di tessuto si trova in ogni pixel dell'immagine. Questa operazione è necessaria poichè ci permetterà tramite operazioni successive di 'filtrare' solo i tessuti che ci interessano scartando tutti gli altri. Questo si fa attraverso la formula specificata in (5.3).

$$HU = (pixel_value * slope) + intercept \quad (6.1)$$

Dove *slope* e *intercept* sono due valori ottenibili dall'intestazione del relativo file DICOM e rilevati durante la TC. Convertiti in HU i valori di tutti i pixel dell'immagine, si può procedere con l'operazione successiva. TC diverse ottenute da macchinari diversi avranno una diversa calibrazione del dispositivo che risulta in leggere variazioni nel modo in cui vengono salvati i dati, come ad esempio lo spessore di una singola slice (indicata nell'header DICOM come *Slice Thickness*) e lo spazio che intercorre tra un voxel e l'altro (indicato come *Pixel Spacing*). In particolare, questo risulta in diversi macchinari aventi voxel di dimensioni differenti: le TC ottenute tramite

un dispositivo A potrebbero avere un voxel di 0.8 mm, mentre un dispositivo B con una diversa calibrazione avrà un voxel di 0.7 mm. Per permettere un confronto univoco tra TC provenienti da macchinari diversi, è quindi necessario eseguire una operazione detta *resampling*, in cui tutte le slice di tutte le TC vengono ricampionate per adattarsi ad una dimensione specificata, generalmente un voxel di dimensione 1x1x1 mm. Una volta effettuata la conversione in HU dei pixel di tutte le immagini di una TC ed un resampling, è possibile passare alla fase di segmentazione vera e propria dell'immagine. Innanzitutto, cosa vuol dire fare segmentazione di una immagine?

Nell'ambito dell'elaborazione digitale delle immagini la segmentazione è la suddivisione di una immagine in regioni significative, andando a classificare i pixel con caratteristiche comuni come colore o intensità. Avendo effettuato la conversione in scala HU dei valori di pixel delle immagini, possiamo quindi suddividere la nostra immagine in regioni in base al tipo di tessuto rappresentato da ogni pixel ed eliminare ciò che non ci interessa.

6.1 Preprocessing dell'immagine

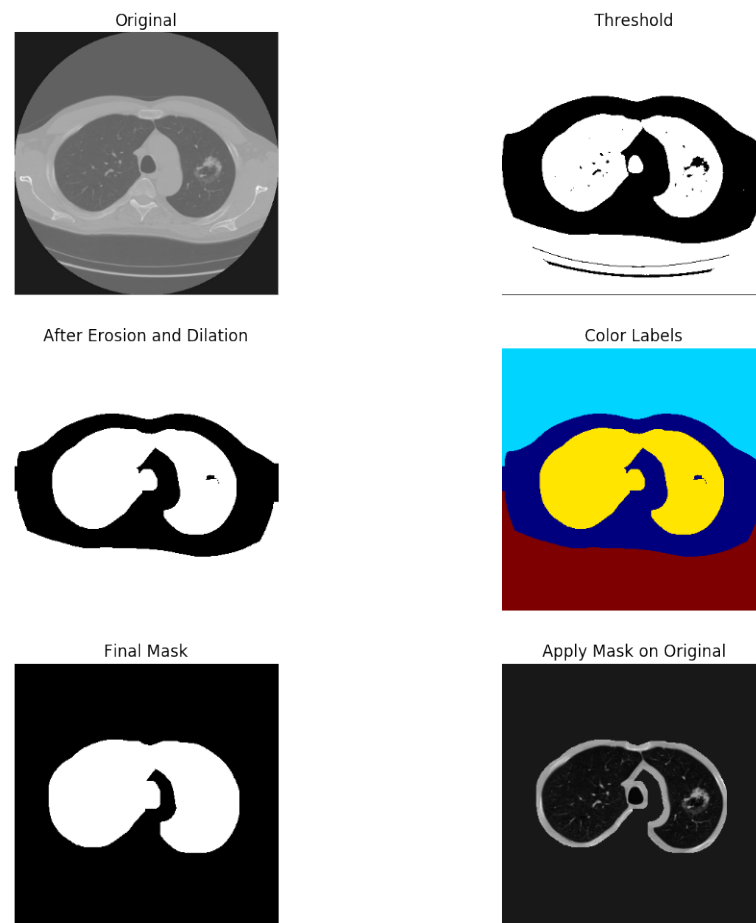


Figura 6.2: Esempio di generazione maschera e maschera applicata su immagine originale

Per fare ciò è necessario ottenere delle "maschere di segmentazione" che vanno poi applicate all'immagine di partenza. Queste maschere si ottengono innanzitutto attraverso un clustering K-means che va ad individuare il valore di soglia che permette di distinguere le zone dell'immagine in cui sono presenti tessuti molli od ossa da quelle che contengono i polmoni od aria (ciò che ci interessa). In seguito attraverso operazioni di digital image processing come erosioni e dilatazioni si vanno prima ad eliminare piccoli elementi come pixel sparsi che possono costituire rumore e poi a includere nuovamente nella zona identificata come polmoni alcuni dei pixel attorno ai polmoni, in modo da non rischiare di tagliarli accidentalmente al di fuori dell'immagine. Ad ognuna di queste zone individuate viene assegnata una etichetta e la maschera corrispondente all'etichetta "polmoni" viene applicata sull'immagine originale attraverso una operazione di moltiplicazione tra matrici. Un esempio di

tutte le fasi della segmentazione è visibile in 6.2.

Possiamo vedere come la segmentazione ha eliminato tutto ciò che non ci interessava dall'immagine di partenza, ovvero ogni tessuto che non fosse il polmone. Questa potrebbe essere vista come il termine dell'operazione di preprocessing dell'immagine, ma si potrebbe pensare di procedere ulteriormente. Allo scopo di individuare due tipologie diverse di tumore infatti, non ci interessa andare ad esaminare l'intero polmone ma solo una sua piccola parte in ogni immagine, che quindi contiene ancora ulteriore rumore. La parte che ci interessa sono i vasi sanguigni, dal momento che sono la zona polmonare dove tendono a formarsi i noduli delle due tipologie di tumore in questione. Possiamo quindi utilizzare di nuovo la scala HU per andare a rimuovere dall'immagine tutti quei pixel con valori che non corrispondono ai vasi sanguigni, come è visibile in figura 6.3.

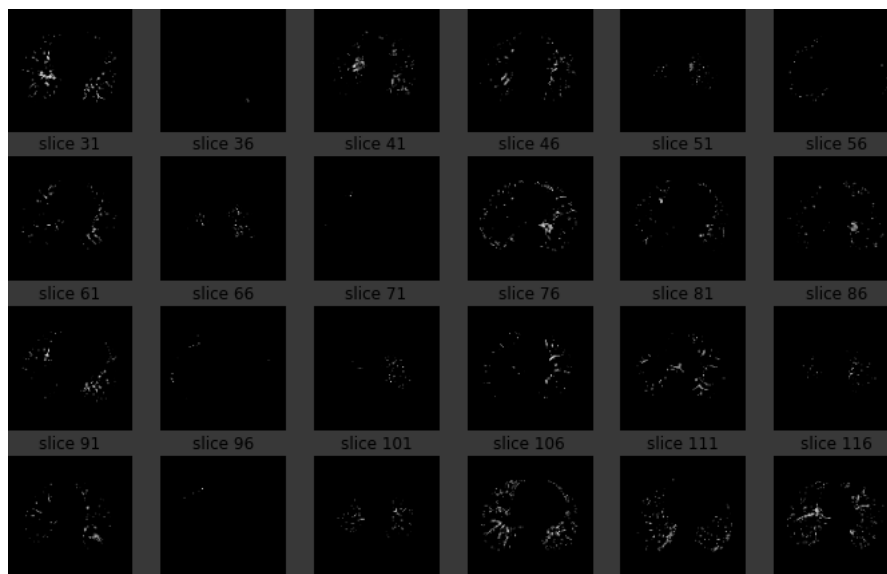


Figura 6.3: Un esempio di ulteriore filtraggio dei vasi sanguigni su diverse slice di una TC

Infine, come ultimo passo della sequenza di preprocessing, i valori di tutte le slice di una TC vengono normalizzati in modo da essere compresi tra 0 ed 1. La figura 6.4 riassume i vari passi del preprocessing completo di una immagine DICOM.

6.1 Preprocessing dell'immagine

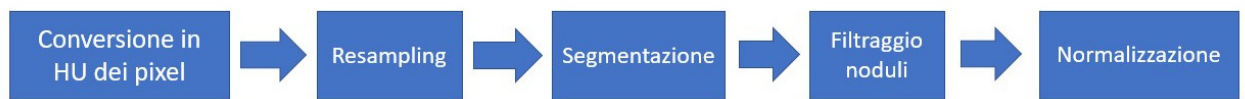


Figura 6.4: Riassunto del procedimento di preprocessing di una immagine

Capitolo 7

Esperimenti effettuati

Prima di entrare nel dettaglio dei vari esperimenti effettuati durante la stesura di questa tesi, è bene, dopo aver introdotto tutti i concetti teorici utilizzati come il funzionamento di una CNN, fare anche una breve panoramica delle tecnologie e degli strumenti utilizzati, sia per chiarezza che per rendere più semplice la riproduzione degli esperimenti effettuati a scopo di test.

7.1 Il progetto Jupyter

Il Progetto Jupyter [22] è un progetto no-profit e opensource che punta al creare un ambiente che facilita lo sviluppo e la condivisione di codice. In particolare, i Notebook Jupyter forniscono un ambiente di sviluppo integrato in un browser web, che permette di scrivere, editare e commentare codice in blocchi modulari che possono essere eseguiti singolarmente direttamente nel browser, invece di eseguire per intero tutto il codice. Supporta inoltre svariati linguaggi di mark-up come HTML e LaTeX per una presentazione e visualizzazione più scorrevole dei dati.

7.1.1 Google Colab

Google Colab [23] invece ospita dei Notebook Jupyter sui propri server mettendo a disposizione di chiunque (con determinate limitazioni) accesso gratuito a GPU per un massimo di 12 ore consecutive. I notebook Colab vengono salvati sul proprio account Google Drive e sono accessibili ed eseguibili da chiunque ha il permesso di accedervi, facilitando la condivisione del codice. Un notebook Colab di fatto non è quindi altro

che un notebook Jupyter, e proprio per questo, anzichè collegarsi ai server Google, è possibile anche collegare un notebook Colab alla propria macchina in locale avviando una sessione Jupyter sulla propria macchina. Di questa funzionalità parleremo più approfonditamente quando andremo ad illustrare gli esperimenti effettuati sul modulo Jetson TX2. Nella versione gratuita un notebook Colab mette a disposizione una CPU da 12 GB di RAM e 97 GB di spazio su disco sull'istanza della virtual machine a cui ci si collega (di cui 31 GB già occupati da tutto l'ambiente Colab e le librerie già pre-installate) e anche una GPU da 12 GB di RAM e 60 GB di spazio su disco (di cui sempre 31 GB già occupati). Il limite di una istanza per la versione gratuita è un runtime di massimo 12 ore consecutive, dopodichè si viene disconnessi e tutte le variabili create in locale perse. Si viene inoltre disconnessi dalla istanza Colab "per inattività" se non viene rilevato nessun comando in esecuzione per più di 15 minuti, in modo da minimizzare lo spreco di risorse da suddividere tra tutti gli utenti che ne fanno richiesta. Recentemente, Colab ha iniziato a mettere a disposizione anche delle TPU. Utilizzando un account Colab Pro è possibile mantenere attiva un'istanza fino a 48 ore consecutive e si hanno accesso a maggiori risorse: 225 GB di spazio su disco per la CPU con 12 GB di RAM, 147 GB di spazio su disco per la GPU e sempre 12 GB di RAM e 225 GB di spazio su disco per la TPU con 33 GB di RAM, oltre ad una maggiore "priorità" nell'accesso alle risorse richieste e un tempo maggiore necessario prima di venire disconnessi per inattività (circa 2-3 ore dagli esperimenti effettuati). Una delle maggiori attrattive di Google Colab (oltre all'ovvia disponibilità gratis di una GPU e TPU) che lo rendono così utilizzato è l'intero ambiente integrato e l'insieme di librerie pre-installate che ne permettono l'utilizzo in ambiti come analisi dei dati e ML in modalità "out-of-the-box", ovvero pronto per essere utilizzato senza dover configurare un intero ambiente di sviluppo. L'unico pre-requisito all'utilizzo dei notebook Colab è l'essere in possesso di un account Google Drive.

7.2 Linguaggi e librerie utilizzate

Colab permette di scrivere ed eseguire codice in una moltitudine di linguaggi: principalmente codice Python, ma anche comandi come script bash e comandi di sistema.

Python [24] è un linguaggio di programmazione di alto livello orientato agli oggetti (anche se non completamente basati come ad esempio Java) estremamente utilizzato nell'ambito della computazione numerica, analisi di dati, applicazioni distribuite e scripting. Python è continuamente aggiornato e presenta tantissime librerie che lo rendono adatto agli usi più vari. È stato pensato per essere facilmente leggibile e comprensibile; tra le sue caratteristiche più inusuali ad esempio troviamo il metodo utilizzato per suddividere i vari blocchi di un programma. Anziché le parentesi graffe `{ }` o altri simboli per delimitare funzioni o il simbolo `;` al termine di ogni istruzione usati da altri linguaggi Python richiede unicamente che il codice sia correttamente indentato, attraverso l'uso del carattere stesso di indentazione o di un numero coerente (e corretto) di spazio. Questo obbliga per necessità a scrivere codice correttamente indentato, rendendolo immediatamente più leggibile e di facile interpretazione. Tutto il codice utilizzato in questa tesi è stato scritto principalmente in Python.

NumPy [25] è una libreria open-source scritta per il linguaggio di programmazione Python (da lì il nome della libreria) ed è una delle librerie fondamentali e più utilizzate per la computazione scientifica di dati in Python. Alla base della libreria NumPy abbiamo l'oggetto *ndarray*, che permette di rappresentare array multidimensionali di dati omogenei (ovvero tutti dello stesso tipo, ad esempio interi o float), e la libreria offre svariate funzioni per effettuare operazioni rapide su questi *ndarray* come operazioni matematiche, logiche, di confronto e di ordinamento e altre come operazioni di algebra lineare e trasformata di Fourier. Nell'ambito del ML e soprattutto dell'analisi delle immagini, NumPy è praticamente universalmente utilizzato per la rappresentazione di immagini attraverso matrici $n \times m$, dove $n \times m$ sono le dimensioni dell'immagine, e di molte altre tipologie di dati.

Keras [26] è un'altra libreria open-source scritta per Python appositamente per il ML e le reti neurali. Lo scopo di Keras è di fornire una interfaccia chiara e facilmente utilizzabile da un operatore umano per la creazione e lo sviluppo di reti neurali profonde, fornendo API chiare e concise e cercando di minimizzare il numero di operazioni richieste allo sviluppo di una rete neurale, dalla sua creazione al suo addestramento fino alla fase finale di testing e fine-tuning dei parametri. Supporta come back-end le librerie TensorFlow, Microsoft Cognitive Toolkit e Theano. Dal 2017 Keras è ufficialmente supportata dal team di TensorFlow. TensorFlow [27] è

una libreria e piattaforma open-source per il ML, sviluppata da Google Brain e resa disponibile a partire dal 9 novembre 2015 sotto la licenza open source Apache 2.0. TensorFlow non è scritta appositamente per il linguaggio Python, ma fornisce API native anche per altri linguaggi come C/C++, Java e Go ed è compatibile con un gran numero di dispositivi, dai principali sistemi operativi a 64 bit (Windows, MacOS X e Linux) ai sistemi Android, quindi anche su dispositivi mobile e Internet of Things attraverso TensorFlow Lite. Scikit-learn [28] è una libreria open source per il ML scritta per il linguaggio Python. Implementa tra le altre cose algoritmi di classificazione, regressione, clustering e macchine a vettori di supporto (Support Vector Machine, SVM), ma anche funzioni per la valutazione della validità di un modello che vanno a calcolare le varie metriche illustrate in precedenza nel capitolo 4 come accuratezza, F1 score e AUC score. E' progettato per lavorare in congiunzione alla libreria NumPy. Scikit-image [29] è un'altra libreria open source scritta per il linguaggio Python che contiene un insieme di algoritmi utili ad effettuare operazioni di processamento e manipolazione delle immagini. Pydicom [30] è infine una libreria scritta per il linguaggio Python che permette di aprire, leggere, modificare e scrivere file in formato DICOM attraverso semplici comandi Python. Anche qui viene spesso utilizzata in congiunzione alla libreria NumPy per trasformare il contenuto di una immagine DICOM in un oggetto *ndarray* manipolabile poi come qualsiasi altro oggetto dello stesso tipo. Infine, per visualizzare risultati che necessitavano di grafici (come le curve ROC) si è fatto uso della libreria Matplotlib [31]. Tutte le librerie appena illustrate (e moltissime altre) ad eccezione di pydicom sono già pre-installate e pronte per l'uso nell'ambiente Colab oltre ad essere automaticamente e costantemente aggiornate alla loro versione stabile più recente. E' comunque possibile, se se ne ha bisogno, installare qualsivoglia altra libreria (o versione precedente di una libreria già installata) semplicemente eseguendo il relativo comando python in un notebook Colab e si avrà a disposizione la libreria installata per tutta la durata del runtime di quella specifica istanza.

7.3 Dataset utilizzato

Il dataset utilizzato durante gli esperimenti effettuati in questa tesi è formato da un insieme di TC raccolte nel corso degli anni in diversi ospedali e diversi macchinari, resi disponibili attraverso vari dataset pubblici reperibili online. Dalla piattaforma Cancer Imaging Archive sono stati utilizzati i seguenti dataset: ‘NSCLC Radiogenomics’, ‘NSCLC Radiomics’, ‘NSCLC Radiomics Genomics’, ‘TCGA LUAD’, ‘TCGA LUSC’, ‘LIDC IDRI’ più altre TC reperite dalla piattaforma Give a Scan. Innanzitutto è stato necessario riordinarlo suddividendo le varie cartelle e sottocartelle nelle due categorie di tumore che siamo andati ad esaminare, l’adenocarcinoma e il carcinoma squamoso. Il dataset è formato in totale da un insieme di 781 TC, di cui 455 di tipologia adenocarcinoma e le restanti 326 di carcinoma squamoso. Per venire dati in input alla rete le due tipologie di file sono stati mischiati attraverso l’utilizzo della funzione *shuffle* della libreria scikit-learn, che ha in questo modo randomizzato l’ordine con cui le TC vengono date in input alla rete. Settando il parametro della funzione *shuffle* "random_state" ad un numero intero (il cosiddetto seed) è possibile richiamare la funzione *shuffle* quante volte si vuole ed essere sicuri di ottenere sempre la stessa lista di file riordinata. Questo torna utile allo scopo di riprodurre gli esperimenti o per confrontare i risultati ottenuti in test che necessitano preprocessing diverso sui dati, in modo da essere sicuri di star dando alle varie reti sempre gli stessi dati e che i risultati diversi ottenuti non siano dovuti semplicemente ad un diverso ordine dei file in input. Altro particolare importante è la suddivisione del dataset. Come menzionato nel capitolo 2 sulla teoria dell’apprendimento, nel cosiddetto apprendimento supervisionato si danno in input alla rete dei cosiddetti dati di training, su cui la rete verrà addestrata allo scopo di minimizzare la funzione di loss e massimizzare l’accuratezza, e dei dati di testing, ovvero dati che la rete non ha mai visto prima per valutare quanto è in grado di applicare ciò che ha imparato (ovvero di generalizzare). Si possono inoltre anche generare dei file di validation, ovvero file che verranno utilizzati per una prima valutazione della rete direttamente durante la fase di training (ma su cui la rete non si addestra). Anche la suddivisione del dataset è stata fatta utilizzando una funzione della libreria scikit-learn, chiamata *train_test_split*. Alla funzione sono stati dati in in input la lista di file mischiati (con le relative label associate) e la percentuale in cui vogliamo suddividere tutta la

lista in file di train e testing. Impostando il parametro `test_split` a 0.5, ad esempio, si sarebbero ottenute in output due liste dove la metà di tutti i 781 file vanno a formare la lista di training e l'altra metà quella di testing. Nel nostro caso, per creare un set di training, uno di validation e uno di testing, la funzione è stata richiamata due volte. La prima volta con un `test_split` di 0.2 (20%) per suddividere tutti i file in due gruppi da 624 (train e validation) e 157 file (testing) e una seconda volta sempre con un `test_split` di 0.2 per risuddividere i 624 file in 499 che verranno usati effettivamente per il training e 125 per la validation.

7.4 Rete C3DKeras

Uno dei due modelli di rete convoluzionale 3D che è stato utilizzato per parte degli esperimenti di questa tesi è chiamato C3D for Keras. Come intuibile dal nome si tratta di un adattamento per renderlo compatibile con Keras di un modello di rete 3D convoluzionale sviluppato originariamente per Caffe nel paper di Du et al (2015)[32]. E' a sua volta una modifica del modello BVLC caffe [33] che è stato addestrato sul dataset Sports-1M che contiene clip di video di vari sport allo scopo di riconoscere la tipologia di sport contenuta in ogni clip video. Ciò che ci interessava vedere era se, utilizzando la rete pre-addestrata con i pesi già aggiornati messa a disposizione, fosse possibile fare uso del transfer learning per riaddestrare solo il classificatore (ovvero i layer finali fully connected della rete) ad un nuovo scopo. Per fare ciò tuttavia oltre alle operazioni di pre-processing illustrate nel capitolo 5 erano necessarie un altro paio di operazioni sul dataset. C3D è infatti stata addestrata a prendere in input clip video di 16 frame, dove ogni frame è costituito da una immagine rgb di 112x112 pixel. Era quindi necessario adattare le nostre TC segmentate a queste dimensioni. Se per la dimensione della singola immagine basta effettuare un ridimensionamento, per ottenere gruppi di 16 slice per ogni TC si è dovuto fare ricorso alla cosiddetta tecnica dello zero padding, che opera come segue:

Si vanno ad esaminare il numero di slice della singola TC. Se il numero di slice è divisibile per 16, si suddivide la slice in n gruppi da 16 slice, ad ognuno dei quali viene poi associata la label che corrispondeva alla TC intera. Se il numero di slice non è divisibile per 16, ovvero se il numero di slice diviso 16 restituisce un resto

m diverso da 0, si sottrae a 16 il numero m ottenuto e si crea un ulteriore gruppo da 16 contenente le m slice più un numero di slice artificiali riempite di soli zeri (che corrispondono a immagini completamente nere) necessarie a raggiungere le 16 slice. Il feature extractor della rete è costituita da 5 layer di convoluzione 3D a cui si vanno ad alternare altrettanto livelli di MaxPooling3D, in modo da ridurre la quantità di dati da elaborare cercando tuttavia di perdere meno informazione possibile. A quel punto tramite un layer denominato Flatten si va a, come suggerisce il nome, appiattare tutte le feature ottenute in un unico grande vettore di dimensioni 8196×1 . A questo punto nel modello originale seguivano i layer fully connected che effettuavano la classificazione vera e propria. Trattandosi di un problema di classificazione multiclasse (riconoscere tra 487 tipologie di sport diversi), è stato necessario modificare proprio questa ultima parte della rete per adattarla al nostro problema, che è invece un problema di classificazione binaria. E' possibile rimuovere o aggiungere layer ad un modello keras tramite le funzioni `.pop()`, che rimuovono l'ultimo layer che trovano a partire dal "basso" e `.add()`, che permette di aggiungere layer di diverso tipo specificandone tipologia (nel caso di layer fully connected, si tratterà di layer denominati Dense), numero di nodi, funzione di attivazione ed eventualmente altri parametri. Nel nostro caso abbiamo ridotto il numero di nodi dei layer fully connected, non avendo bisogno di così tanti parametri come nel caso di un problema di classificazione multiclasse, e soprattutto ridotto il numero del layer di output a 2 con funzione di attivazione softmax, in modo da ottenere in output dalla rete la probabilità di un campione di appartenere alle due classi da valutare. Altro particolare importante quando si fa uso del transfer learning è appunto l'assicurarsi di non andare a modificare i pesi della rete pre-addestrata, nel nostro caso la sequenza di layer di convoluzione e di maxpooling 3D, altrimenti l'addestramento della rete viene rifeffettuato da zero e le feature apprese in precedenza perse. Per fare ciò è possibile "congelare" determinati layer, in modo che in fase di training i loro pesi non vengano aggiornati. In Keras questo si ottiene impostando il parametro "trainable" di un layer al valore "false", in quanto di default Keras assume che tutti i layer di una rete vadano addestrati. In questo modo è possibile caricare nel modello dei pesi ottenuti da un addestramento precedente ed essere certi che quei pesi non verranno mai modificati, mentre durante il training si andranno ad aggiornare i pesi dei nuovi

layer da addestrare per il nuovo compito da svolgere. Provenendo tuttavia il modello base della rete C3D da un framework diverso da Keras che memorizza i pesi in un formato leggermente diverso, è tuttavia prima necessario eseguire uno script che converta i pesi nel formato accettato da Keras. A quel punto è possibile eseguire il caricamento dei pesi tramite il comando integrato `.load_weights()` ed iniziare normalmente l'addestramento come in un qualsiasi modello Keras.

7.5 Rete ConvLSTM

A differenza della particolare rete C3D che doveva prima essere adattata a diventare compatibile con Keras, è possibile definire direttamente attraverso la libreria dei layer ConvLSTM. Come illustrato nel capitolo 3 le ConvLSTM combinano le operazioni di convoluzione 3D che estraggono feature da una sequenza di immagini con una "cella" di memoria che stabilisce quali informazioni lungo la sequenza sono le più importanti da memorizzare. Inoltre, non facendo uso del transfer learning, non è stato necessario sottostare alle limitazioni di C3DKeras riguardanti le dimensioni dei dati in input, in quanto la rete viene addestrata da zero. Essendo incluso nella libreria Keras, basta semplicemente importare un layer ConvLSTM come qualsiasi altro layer e definire alcuni parametri essenziali per il suo funzionamento in modo simile ai semplici layer convoluzionali 3D. ConvLSTM accetta in input tensori a cinque dimensioni; in base al valore del parametro "data_format" che può assumere i valori "channels first" o "channels last" gli input accettati dovranno essere rispettivamente del formato (samples, time, channels, rows, cols) oppure (samples, time, rows, cols, channels) dove samples è il numero totale di campioni da analizzare in input, time è il numero di elementi di una sequenza (o nel nostro caso, trattandosi di sequenze analizzate lungo l'asse spaziale e non temporale, il numero di slices della TC da analizzare), channels il numero di canali della singola immagine (nel nostro caso 3 per una immagine RGB. Avremmo potuto utilizzare anche 1 solo canale dato che la TC è una immagine in scala di grigi, ma abbiamo utilizzato anche qui 3 canali per mantenere una certa conformità con la C3DKeras) e rows e cols sono rispettivamente le dimensioni della singola immagine, vista come una matrice con un determinato numero di righe e di colonne. Altro parametro importante è il valore

del parametro "return_sequences", che può essere True o False. Se impostato a True, nell'output dello strato ConvLSTM verranno inclusi oltre ai filtri ottenuti dopo l'operazione di convoluzione (il cui numero è stabilito attraverso il parametro "filters" che prende in input un numero intero), alle nuove righe e nuove colonne delle immagini anche i timesteps della sequenza, che può essere utile se si vogliono concatenare in sequenza più di un layer ConvLSTM, in quanto i layer successivi al primo prenderanno quindi in input l'informazione sulle sequenze memorizzate nel layer precedente. Se invece come nel nostro caso si ha un solo layer ConvLSTM (o nel caso di layer multipli, l'ultimo layer ConvLSTM), si può impostare il parametro a False e il layer restituirà in output un tensore a 4 dimensioni nel formato (samples, filters, new_rows, new_cols), dove samples è il numero di campioni dati in input, filters contiene il risultato delle feature estratte dall'operazione di convoluzione e new_rows e new_cols sono le nuove dimensioni dell'immagine dopo la convoluzione. A seguire al layer ConvLSTM abbiamo un layer denominato di Dropout. Il layer di Dropout prende in input un parametro detto rate compreso tra 0 ed 1 ed ha il compito di impostare randomicamente un input ricevuto dal layer precedente a 0. L'operazione viene eseguita con una frequenza pari a 0 ad ogni passo del training, ed è una operazione necessaria ad evitare il cosiddetto fenomeno di "overfitting". L'overfitting si verifica quando una rete raggiunge accuratezza altissima sui dati di training, ma unicamente su quelli, e non è invece in grado di applicare ciò che ha imparato su dati che non ha mai visto (ovvero di generalizzare). A seguire al layer di Dropout abbiamo l'operazione di Flatten che come visto già nella rete C3D inserisce tutte le feature estratte in un unico vettore a cui seguono poi due layer fully connected (intervallati a loro volta da Dropout) ed infine un ultimo layer fully connected con 2 nodi e funzione di attivazione softmax per il risultato finale, in modo simile alla rete C3D. Tutti i rate dei layer di Dropout sono stati impostati ad un valore di 0.5, che corrisponde ad una possibilità del 50% della rete di impostare le unità di input a 0 ad ogni passo della fase di training. La rete ConvLSTM utilizzata nei nostri esperimenti è molto meno "profonda" della rete C3D con cui abbiamo voluto confrontarla, ma ciò è compensato dal non essere limitata dalle dimensioni delle immagini da dare in input, in particolare dal poter dare sequenze lunghe più di 16 frame. Lavorando la ConvLSTM meglio con sequenze relativamente corte (inferiori ai 100 frame), abbiamo

deciso di lavorare con sequenze di 80 slice per ogni TC. Ovviamente, la stragrande maggioranza delle TC, soprattutto dopo il resampling, presentava un numero di slice anche molto maggiore ad 80, per cui si sono dovute "tagliare" le TC in modo da prenderne solo una parte. Ovviamente la parte di maggiore interesse per i nostri esperimenti è la parte centrale del polmone, e si è quindi utilizzato un algoritmo per prendere per ogni TC le 80 slice centrali (numero ovviamente arbitrario e che in esperimenti futuri si potrà provare ad aumentare). L'algoritmo procede come segue: Si prende il numero n di slice della TC, che corrisponde alla prima dimensione della TC rappresentata come array NumPy, dopodichè si sottrae 80 a quel numero per determinare il numero di slice che andranno tagliate dalla tac. Se il numero di slice da eliminare è pari, attraverso la funzione slice della libreria NumPy eliminiamo $n/2$ slice dall'inizio e $n/2$ slice dalla fine della TC, in modo da restare con le 80 slice centrali. Se invece il numero di slice da eliminare è dispari, sempre attraverso la funzione slice eliminiamo $n/2$ slice dall'inizio ed $n/2 + 1$ slice dalla fine, in modo da ottenere sempre le 80 slice centrali della TC. Nel (raro) caso in cui la TC di partenza abbia meno di 80 slice, si sottrae il numero di slice attuali a 80 per determinare quante slice "artificiali" occorre aggiungere attraverso il metodo illustrato in precedenza dello zero padding.

7.6 Suddivisione degli esperimenti

Gli esperimenti sono stati suddivisi in base alla tipologia di pre-processing effettuato sulle TC. Si è voluti andare a confrontare le performance della C3D e della ConvLSTM su tre dataset diversi: il primo dataset, sempre suddiviso in training, validation e testing, conteneva le immagini TC a cui era stato applicato unicamente l'operazione di resampling e resize necessario a darlo in input alla rete, senza nessun altro tipo di pre-processing applicato. In altre parole, si sono andate a valutare le performance delle due reti con immagini più "pure" e meno modificate possibili. Il secondo dataset, con la stessa suddivisione, era formato dalle TC a cui era stato applicato il preprocessing completo, ovvero resampling, crop e segmentazione del polmone, rimozione delle maschere di segmentazione incorrette con immagini completamente nere e ulteriore filtraggio per mantenere unicamente i noduli ed i vasi sanguigni della slice. Il terzo dataset, sempre con la stessa suddivisione, è formato dalle

TC a cui è stato applicato lo stesso procedimento di preprocessing senza però l'ulteriore filtraggio di noduli e vasi sanguigni. Per tutti gli esperimenti effettuati l'addestramento della rete è stato fatto partire specificando un massimo di 100 epoche, che tuttavia sono spesso fin troppe per addestrare un modello senza incappare in un fenomeno di estremo overfitting. L'ideale è, ovviamente, arrestare l'addestramento della rete quando si raggiunge la performance migliore della rete in termini di accuratezza e minimizzazione della funzione di loss. Keras permette di fare questo attraverso l'uso di Callbacks. Un Callback di Keras è un oggetto che permette di eseguire una determinata funzione in determinati punti dell'addestramento della rete, ad esempio all'inizio o alla fine di una epoca, prima o dopo un singolo batch di dati e così via. Keras mette a disposizione svariate API già pronte, ma permette anche di definire i propri callbacks personalizzati come estensioni della classe Callbacks. In particolare, le API di nostro interesse utilizzate durante i nostri esperimenti si chiamano ModelCheckpoint, EarlyStopping e CSVLogger. ModelCheckpoint permette, al rispettare di determinate condizioni, di salvare periodicamente i pesi della rete durante il training in un percorso specificato, in modo da non perdere i risultati ottenuti fino a quel momento se questo viene interrotto. Specificando un percorso diverso ad ogni epoca (ad esempio con una variabile i che viene incrementata alla fine di ogni epoca e aggiunta al nome del file per crearne uno diverso) è possibile in teoria riprendere l'addestramento da qualsiasi epoca, ma è una modalità che tende ad occupare molto spazio e non essere comunque ottimale rispetto a quella adottata in questa tesi. Ponendo il parametro "save_weights_only" a False è possibile salvare non solo i pesi ma anche l'intero modello, che include struttura dei layer, funzione di loss e ottimizzatore utilizzati ed è essenziale se si vuole riprendere l'addestramento della rete in un momento successivo. Ponendo invece a True il parametro "save_best_only" il modello non verrà salvato alla fine di ogni epoca indipendentemente dalla performance ma solo se è migliorato rispetto all'ultimo modello salvato (chiaramente, alla prima epoca il modello verrà salvato in ogni caso). Se il modello è migliorato o meno viene definito dal nome della variabile passata al parametro "monitor", che decide appunto in base all'andamento di quale variabile durante il training decidere se salvare l'attuale modello sovrascrivendo il risultato precedente. Monitor può assumere quattro valori: loss, accuracy, val_loss e val_accuracy. Chiaramente se si decidono di monitorare le

funzioni di loss, il modello verrà salvato solo se la loss è diminuita rispetto al valore precedente. Se invece si monitorano le accuratèzze, il modello viene salvato solo se l'accuratèzza è aumentata. In tutti i test effettuati i modelli sono stati monitorati attraverso la `val_accuracy`. Questo perchè andando a monitorare la loss e la accuracy di training si possono andare a salvare piu' facilmente modelli che stanno overfittando, in quanto questo fenomeno è spesso contraddistinto proprio dalla loss e accuracy di training che continuano a diminuire e aumentare rispettivamente mentre altrettanto non fanno le `val_loss` e `val_accuracy`, che invece spesso peggiorano in presenza di overfitting. Per ridurre quindi il rischio di salvare modelli che presentavano overfitting estremo si è deciso di monitorare direttamente la accuratezza del dataset di validation. Il secondo callback utilizzato è `EarlyStopping`, che come suggerito dal nome è ciò che permette effettivamente di interrompere il training una volta che un parametro monitorato ha smesso di migliorare. Anche qui abbiamo utilizzato come parametro di valutazione la `val_accuracy`, per gli stessi motivi elencati prima. Parametro importante oltre alla variabile da monitorare è il parametro "patience", che prende in input un numero intero e stabilisce la "pazienza" del monitor prima di interrompere il training. Questo perchè spesso può accadere che loss e accuratezza hanno per qualche epoca un andamento altalenante prima di stabilizzarsi, perciò interrompere l'addestramento di una rete non appena il parametro valutato smette di migliorare può essere controproducente. Il numero intero impostato come "pazienza" è quindi il numero di epoche di fila dove il parametro monitorato non migliora che dovranno passare prima che l'addestramento venga effettivamente interrotto, in modo da dare tempo se necessario alla loss o alla accuratezza di "assestarsi". Questo numero varia enormemente a seconda del tipo di rete utilizzato e del problema che devono risolvere: reti che convergono molto lentamente o con un learning rate molto basso possono richiedere un parametro pazienza anche nell'ordine delle decine di epoche. Per i nostri test si è usato un parametro di pazienza di 7. Il terzo callback utilizzato è `CSVLogger`, che permette di salvare i risultati del training come i valori di loss e accuratezza alla fine di ogni epoca in un file di log, generalmente appunto un csv. Prende come parametri il percorso e nome del file da salvare, il carattere da utilizzare come separatore dei vari campi in un file csv (nel nostro caso, come nella maggior parte dei csv, si è usato il carattere `;`) e il parametro "append" che può

assumere valore True o False. Se False, il file verrà completamente sovrascritto se già esistente, se True, i valori verranno invece salvati in un nuovo file se il nome del file dato come percorso in output non esiste, o inseriti in coda al file esistente. E' utile per continuare a trascrivere i risultati dell'addestramento se questo viene ripreso in un secondo momento, in quanto i valori precedenti di loss, accuracy, val_loss e val_accuracy (la cosiddetta "history" del modello) non vengono altrimenti salvati tra una esecuzione del programma e l'altra. Avere una trascrizione completa dei valori di loss e accuracy durante l'addestramento può tornare utile ad esempio per visualizzare in un grafo tramite matplotlib l'andamento dei due parametri allo scopo di analisi e valutazione del training durante le varie epoche. Dopo aver illustrato come utilizzare i callbacks per monitorare il training, vediamo come si effettua l'addestramento vero e proprio. In Keras l'addestramento di una rete si effettua richiamando il metodo `.fit()`, a cui vanno passati una serie di parametri. I primi sono ovviamente i dati su cui va effettuato il training con le rispettive label, solitamente in formato di due array numpy contenente tutti i dati e le label rispettivamente. Passare i dati alla funzione in questo modo, tuttavia, fa assumere a Keras che i dati vengano caricati tutti insieme nella memoria RAM del dispositivo, che per dataset di dimensioni di svariate decine di Gigabyte spesso non è possibile. Per questo è possibile passare il dataset anche sotto forma di "generatore", un oggetto di Keras che carica i dati in memoria un poco alla volta a gruppi di elementi pari al `batch_size` selezionato, in modo da poter accedere a dataset molto grandi senza aver bisogno di tantissima RAM in grado di caricarlo in memoria tutto in una volta. Keras mette a disposizione delle API già pronte per generare svariati tipi di dati, ad esempio il metodo `.flow_from_directory` se si stanno caricando gruppi di file immagine direttamente da una cartella. Visto che il nostro dataset era rappresentato da array numpy salvati in dataset compressi in formato hdf5 per permettervi un accesso più rapido, è stato necessario definire un generatore custom che prende in input l'oggetto hdf5, recupera dai rispettivi dataset gli array numpy delle TC e delle rispettive label in numero pari al `batch_size` specificato (se ad esempio è pari ad 1, le TC verranno estratte dal dataset una alla volta) e li fornisce alla rete come tupla (array di immagini, label corrispondente). Altri parametri che la funzione `.fit()` prende in input sono ovviamente il numero di epoche per cui la rete va addestrata (100 nel nostro caso), il `batch_size` (che

usando i generatori viene tuttavia specificato direttamente lì), se riordinare in modo casuale i dati e le rispettive label all'inizio di ogni nuova epoca, in modo che la rete eviti di vedere i dati presentati sempre nello stesso ordine (per evitare overfitting, in quanto starebbe apprendendo semplicemente l'ordine delle risposte da dare e non a valutare i dati in base alle feature estratte), i callbacks definiti in precedenza da utilizzare e l'epoca iniziale da considerare. Se si sta addestrando una rete da zero il parametro `initial_epoch` sarà ovviamente pari a 0 e può anche essere non specificato, ma è essenziale se è necessario riprendere un training precedentemente interrotto ad una determinata epoca, in quanto alcuni parametri del modello come il learning rate dipendono dal numero di epoche per cui la rete è stata addestrata e non specificare il numero di epoca da cui riprendere l'addestramento porterebbe ad una configurazione erronea di questi valori. Una volta ultimato l'addestramento della rete si sono utilizzate alcune funzioni della libreria `sklearn` per calcolare le varie metriche attraverso le quali (oltre ovviamente all'accuratezza raggiunta) abbiamo valutato le performance delle reti. Tutte le metriche più utilizzate come precisione, recall, f1 score e accuratezza media si possono facilmente calcolare attraverso la funzione `.classification_report()` importandola dalla libreria `sklearn.metrics`. E' necessario fornire in input a questa funzione il vettore di risultati desiderati (le label corrette dette anche "ground truth labels") ed il vettore di risultati predetti dal classificatore per una serie di dati (nel nostro caso, i dati di testing). Il vettore di risultati predetti dal classificatore si può ottenere richiamando la funzione `model.predict()` a cui passare i dati (le TC del dataset di testing) di cui predire l'output a seguito dell'addestramento, sempre sotto forma di generatore nel caso di dataset molto grandi. Il risultato predetto sarà tuttavia fornito in termini di probabilità di appartenere ad ognuna delle due classi; ad esempio, un possibile output potrebbe essere un vettore `[0.7 0.3]`, dove per il classificatore la TC in input ha il 70% di probabilità di appartenere alla classe positiva e 30% di appartenere alla classe negativa. Le ground truth labels sono nello stesso formato, ovviamente con vettori corrispondenti a `[1 0]` per la classe positiva e `[0 1]` per la classe negativa. Per utilizzare la funzione `classification_report()` tuttavia i vettori devono essere forniti in formato binario (trattandosi di un problema di classificazione binaria). Si possono facilmente convertire i vettori attraverso la funzione `argmax` della libreria `NumPy`, che restituisce per ogni array in input l'indice

in cui si trova il suo elemento più grande. In questo modo un array del tipo [0.7 0.3] si ritrova convertito in 0 che corrisponde alla classe positiva e così via. Per calcolare l'auc score si è invece fatto uso della funzione `roc_auc_score()` sempre della libreria `sklearn.metrics` che prende in input gli stessi vettori utilizzati da `classification report`. Infine, tramite una funzione definita appositamente che prende in input il false positive rate e true positive rate ottenuti tramite la funzione `roc_score()` si è costruito e salvato grazie alla libreria `matplotlib` un grafico della curva ROC per ogni modello. Siccome metriche come precision, recall e di conseguenza l'F1 score vengono calcolate indipendentemente per ogni classe del classificatore, in tutte le tabelle abbiamo riportato per chiarezza e a scopo di analisi il valore ottenuto per ognuna delle due classi, indicando con C0 la classe denominata positiva dal classificatore (adenocarcinoma) e con C1 la classe denominata negativa (tumore a cellule squamose). Di seguito sono riportati i risultati ottenuti con il primo dataset, con immagini a cui è stato applicato il solo resampling e resize a 112x112 pixel, per la rete C3DKeras e la ConvLSTM.

Rete	Accuratezza	F1 score (C0)	F1 score (C1)	AUC
C3D	60%	0.76	0.03	0.5
ConvLSTM	71%	0.76	0.66	0.70

Tabella 7.1: Risultati resampling e resize a 112x112 pixel

Dopodichè i risultati ottenuti con il secondo dataset, con immagini a cui è stato applicato il preprocessing completo più l'ulteriore filtraggio di noduli e vasi sanguigni, sempre a 112x112.

Rete	Accuratezza	F1 score (C0)	F1 score (C1)	AUC
C3D	62%	0.63	0.62	0.62
ConvLSTM	60%	0.75	0.00	0.5

Tabella 7.2: Risultati preprocessing completo con filtraggio noduli

Visti i risultati di questo secondo dataset, si sono fatti test con il terzo dataset, con immagini a cui non è stato applicato il filtraggio di noduli e vasi sanguigni ma la sola segmentazione del polmone, sempre in formato 112x112.

Rete	Accuratezza	F1 score (C0)	F1 score (C1)	AUC
C3D	54%	0.69	0.12	0.53
ConvLSTM	59%	0.70	0.32	0.57

Tabella 7.3: Risultati con solo resampling e segmentazione

Data la generale performance migliore della ConvLSTM, abbiamo voluto fare ulteriori test provando ad incrementare le dimensioni dell'immagine in input, restando sempre con le 80 slice centrali di ogni TC ma passando da immagini 112x112 pixel a immagini 300x300. Abbiamo continuato a lavorare con le immagini a cui era stato applicato resampling e segmentazione in quanto provare a fornire alla rete le immagini con solo resampling, nonostante i risultati incoraggianti mostrati, avrebbe portato ad immagini e quindi alla creazione di dataset di dimensione ancora maggiore, andando a superare i limiti di spazio disponibile su disco sull'istanza Colab e rendendo quindi necessario questo compromesso. Per fare ciò inoltre anche utilizzando la TPU messa a disposizione da Google Colab con 35 GB di RAM il modello era troppo grande causando il crash del runtime a causa dell'esaurimento di tutta la RAM disponibile. Per ridurre la dimensione del modello e quindi anche il numero di parametri da addestrare della rete si è voluto provare a dimezzare il numero di filtri del layer ConvLSTM della rete, da 64 a 32. Questo porta all'estrazione di meno feature per ogni immagine, ma ci interessava sapere se questo veniva bilanciato dalla dimensione dell'immagine quasi triplicata, che porta ad un aumento del numero di pixel di ogni immagine da elaborare di quasi 10 volte. Questo ha quindi portato anche ad un tempo di training molto più lungo (una media di due ore ad epoca). I risultati incoraggianti ci hanno portato a fare altri test a 16 ed infine 8 filtri, cercando l'equilibrio migliore tra dimensione dell'immagine, tempo di addestramento e numero di filtri e valutando l'aumento o diminuzione delle performance in ognuno dei casi.

Rete	Accuratezza	F1 score (C0)	F1 score (C1)	AUC
ConvLSTM(32 filtri)	74%	0.79	0.66	0.72
ConvLSTM(16 filtri)	70%	0.78	0.56	0.66
ConvLSTM(8 filtri)	67%	0.74	0.56	0.65

Tabella 7.4: Risultati con immagini segmentate a 300x300

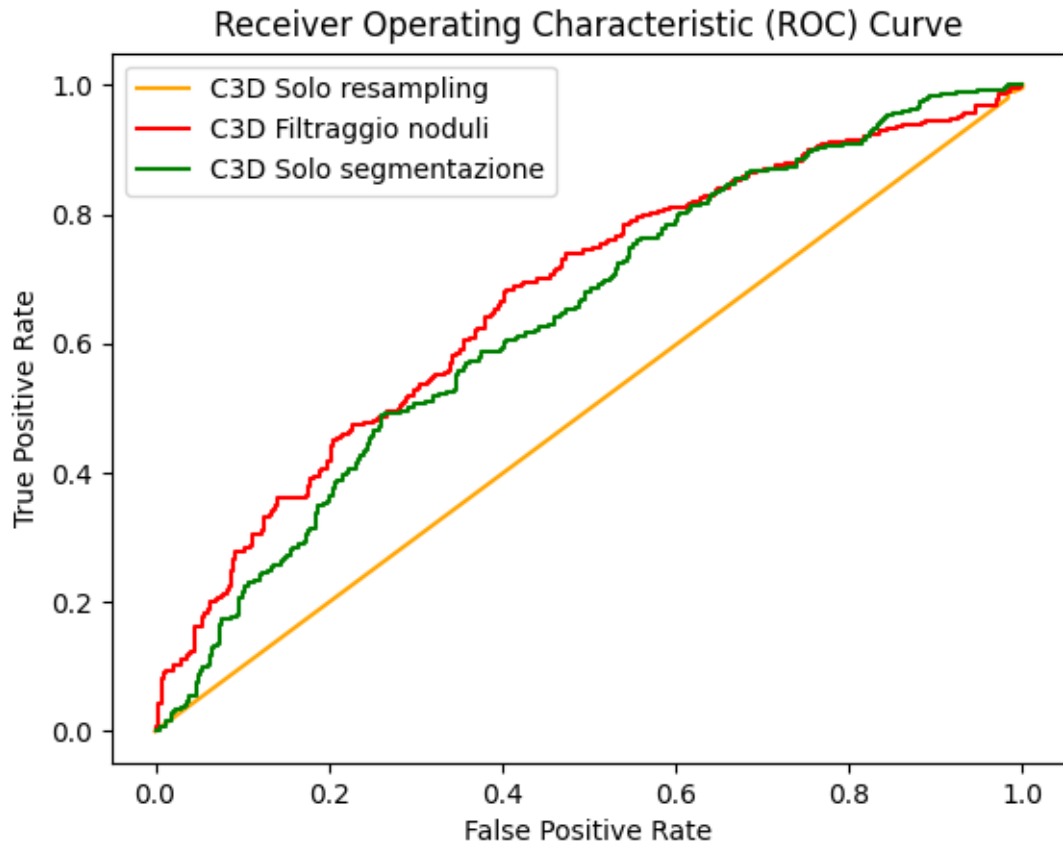


Figura 7.1: Grafo delle curve ROC per i tre esperimenti su C3D

Abbiamo infine riportato il grafo delle curve ROC che mettono a confronto tra di loro la performance generale dei tre esperimenti con la C3D , un altro che mette a confronto le performance sugli stessi tre esperimenti con il modello ConvLSTM ed un grafo che riassume le performance dei tre modelli ConvLSTM che prendevano in input immagini in formato 300x300 pixel . I risultati sono visibili in 7.1,7.2 e 7.3. Abbiamo infine raccolto in un unico grafo tutte le curve ROC per una visione d'insieme, visibile in 7.4.

7.7 Risultati ottenuti

Esaminando i risultati ottenuti dai test effettuati possiamo dedurre alcune considerazioni interessanti: innanzitutto, come illustrato anche nel capitolo 4 di questa tesi, la sola metrica dell'accuratezza non è affatto affidabile come metodo di valutazione

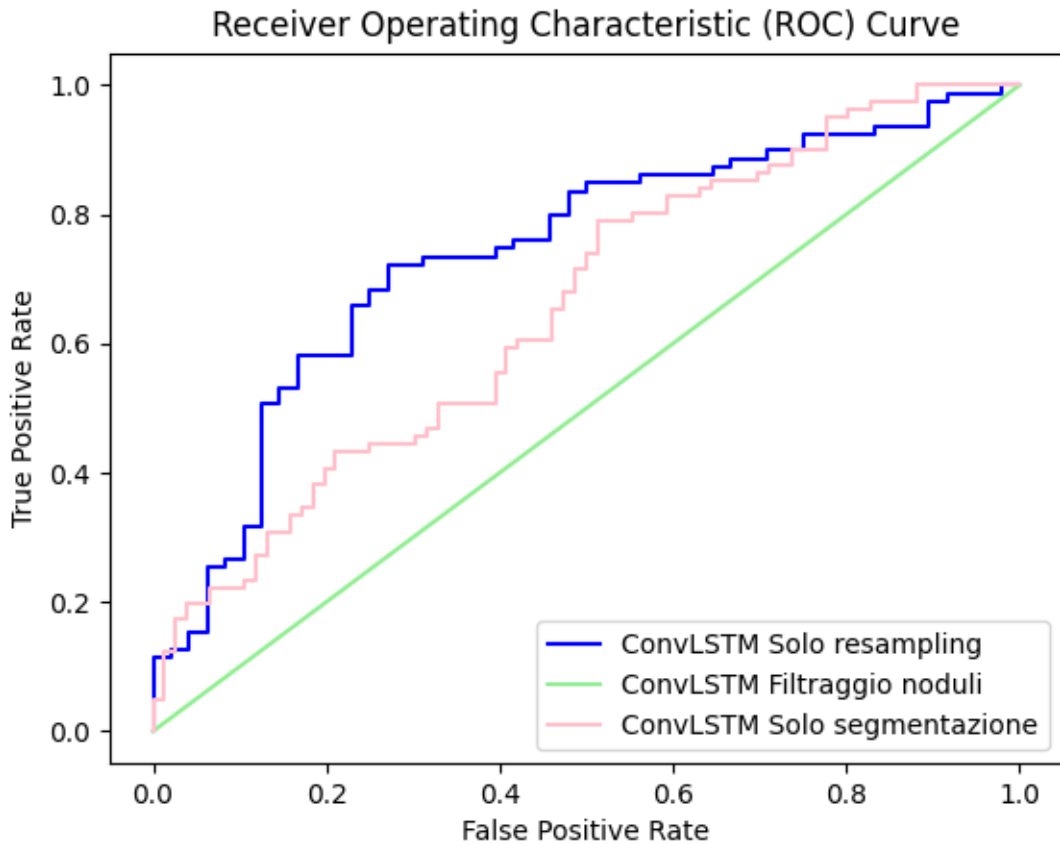


Figura 7.2: Grafo delle curve ROC per i tre esperimenti su ConvLSTM

della performance di una rete. Possiamo vedere che la rete C3D nella Tabella 7.1 sembra avere una accuratezza apparente del 60%, ma andando ad esaminare altre metriche come l’F1 score e l’AUC possiamo vedere che la accuratezza "reale" è in realtà molto più bassa. La rete ha infatti un F1 score per la classe C0 di 0.76, ma addirittura di 0.03 per la classe C1. Questo vuol dire che la rete da come risposta pochissime volte la seconda classe e anche quando lo fa è con scarsa confidenza, sbagliando spesso la predizione. Un altro caso estremo di questo tipo si presenta nella ConvLSTM nella Tabella 7.2 dove alla rete vengono date in input le immagini con ulteriore filtraggio dei noduli. Qui, addirittura, la rete ha un valore completamente pari a 0 nel F1 score della classe C1. Ciò vuol dire in altre parole che la rete in questo caso ha imparato a dare in output sempre e solo la stessa risposta, ovvero la classe C0. Questo comportamento si può spiegare con la scarsa informazione presente in queste immagini: l’ulteriore filtraggio dei noduli rimuove sì tantissimo

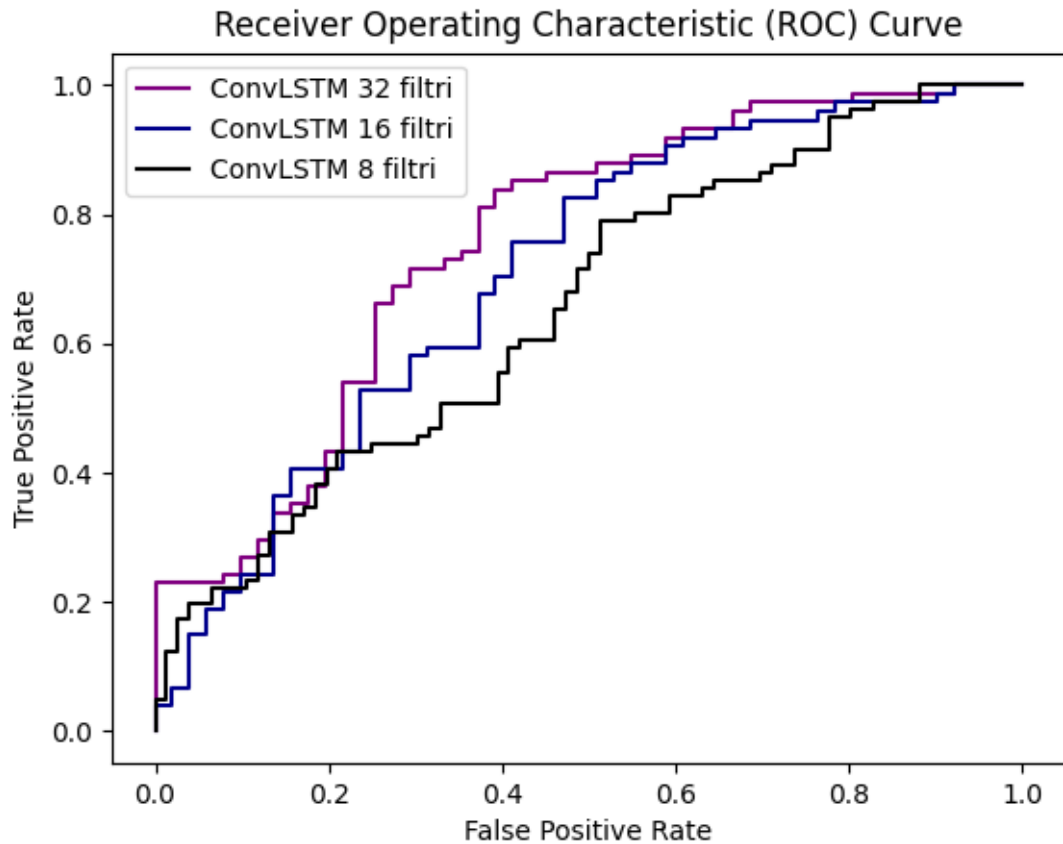


Figura 7.3: Grafo delle curve ROC per i tre esperimenti su ConvLSTM ad immagini più grandi

potenziale rumore dall'immagine, ma evidentemente anche molta informazione e in questo caso la cella di memoria della ConvLSTM non riesce a individuare quale della poca informazione rimasta lungo la sequenza sia importante da memorizzare e quale no, arrivando al comportamento estremo dell'imparare a dare una sola risposta. La C3D in questo caso sembra comportarsi meglio, probabilmente anche grazie al transfer learning, ma non ottiene comunque risultati eccelsi. Con le immagini che presentano la sola segmentazione in Tabella 7.3 vediamo che la C3D peggiora nuovamente: questo si può ipotizzare essere dovuto al fatto che la C3D per la sua architettura intrinseca non considera mai davvero una TC nella sua "interezza", ma sempre scomposta in piccoli gruppi da 16 ognuna con le relative label, e non ha mai quindi una "visione d'insieme" di tutta la TC, al contrario della ConvLSTM, che sembra mostrare una performance che è una esatta via di mezzo tra il solo

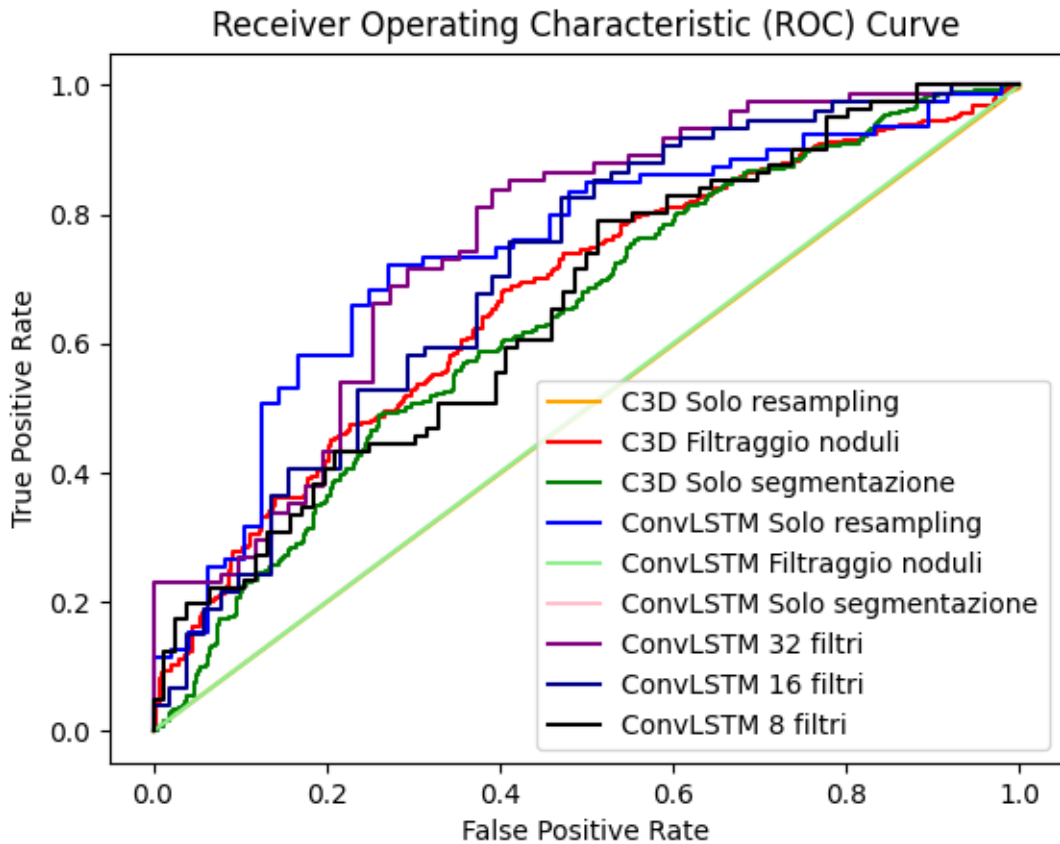


Figura 7.4: Grafo delle curve ROC per tutti gli esperimenti effettuati

resampling e il filtraggio dei noduli, con un AUC leggermente migliore che riflette il leggero aumento nella capacità di distinguere tra le due classi, seppur continuando a prediligere fortemente la classe C0, probabilmente anche a causa di un leggero sbilanciamento del dataset. Non era quindi un risultato completamente inatteso. Infine, aumentando le dimensioni delle immagini in input siamo andati a vedere se era possibile aumentare ulteriormente le performance della rete. Ridurre il numero di filtri (e quindi di features estratte da ogni immagine) sembra avere mostrato risultati molto promettenti venendo bilanciato dalla migliore risoluzione e grandezza delle immagini in input. Come si vede nella Tabella 7.4, è bastato dimezzare il numero di filtri da 64 a 32 per ottenere comunque la performance migliore in assoluto tra tutti gli esperimenti effettuati. 8 filtri porta già ad un abbassamento nella capacità di distinguere tra le due classi ed è probabilmente un numero troppo basso. 16 filtri si colloca approssimativamente ad una via di mezzo tra le due. In ogni caso, la possibilità

di cercare il giusto equilibrio tra complessità dell'immagine e numero di features "discriminanti" da estrarre sembra aver portato a risultati promettenti.

7.8 Esperimenti sul modulo Jetson TX2

7.8.1 Il modulo NVIDIA Jetson TX2

Jetson TX2 [34] è un supercomputer da 7.5 Watt dalle dimensioni ridotte basato sulla GPU della gamma NVIDIA Pascal con 256 core NVIDIA CUDA e pre-caricato con 8 GB di memoria. E' dotato inoltre di uno spazio di archiviazione di 32 GB espandibile tramite microSD. I moduli integrati Jetson della NVIDIA, di cui Jetson TX2 e Jetson TX2i sono tra gli ultimi usciti, si differenziano per dimensioni, potenza di calcolo e robustezza dei componenti, e sono compatibili con un grande numero di interfacce hardware di ogni tipo; le dimensioni ridotte permettono l'installazione su dispositivi come droni o altri moduli come robot industriali o apparecchiature mediche in base alle esigenze.

7.8.2 Il Jetson Developer Kit

Il Jetson Developer Kit [35] è una piattaforma completa per lo sviluppo e la computazione nell'ambito dell'Intelligenza artificiale, ed è particolarmente adatta per applicazioni che richiedono una elevata performance computazionale a potenza ridotta. Il Jetson TX2 Developer Kit fornisce pre-installata su una board la GPU NVIDIA Pascal e svariate interfacce hardware come porte USB 3.0, Bluetooth, HDMI, antenne WiFi e supporto per scheda SD.

7.8.3 Configurazione della Jetson TX2

Prima di poter iniziare ad utilizzare il modulo Jetson TX2 è stato necessario configurarlo per l'uso installando tutti i driver e le librerie necessarie. Per fare ciò è stato necessario l'uso di un computer host su cui è installato Ubuntu 18.04 collegato al modulo Jetson tramite un cavo MicroUSB attraverso il quale la Jetson è stata flashata con l'ultimo firmware disponibile e l'ambiente di sviluppo correlato.

7.8.4 SDK Manager

L'NVIDIA SDK Manager [36] fornisce una piattaforma completa per l'installazione e la configurazione di un modulo Jetson, permettendo di gestire l'installazione di driver, librerie per l'utilizzo in ambiti di intelligenza artificiale, computer vision e simili. L'SDK Manager va installato sul computer host e richiede 8 GB di Ram e una risoluzione dello schermo pari o superiore a 1440x900. Se questi requisiti non sono rispettati (in particolare la risoluzione dello schermo), l'interfaccia grafica del programma risulterà troppo grande e andrà a posizionare i pulsanti che permettono di proseguire con l'installazione 'oltre lo schermo', rendendo impossibile selezionare l'opzione desiderata per proseguire o anche solo scorrere la finestra fino in fondo per leggere tutte le voci disponibili. Fortunatamente, l'SDK Manager offre anche una modalità utilizzabile da riga di comando [37], che ha permesso di proseguire con l'installazione, e permette anche di eseguire il download di tutti i file necessari in anticipo in modo da poter poi proseguire con l'installazione in modalità offline in un secondo momento. L'installazione via riga di comando richiede svariati parametri obbligatori da fornire quando si esegue l'istruzione, andiamo a illustrare brevemente i principali che si sono resi necessari: `-user user_email` permette di specificare l'account da sviluppatore NVIDIA con il quale ci si è registrati alla piattaforma, `-offline` permette di specificare di andare ad utilizzare i file scaricati in precedenza (e relativo percorso a cui trovarli con il parametro `-downloadfolder`). `-cli install | uninstall | downloadonly` permette di specificare l'azione da eseguire, ovvero installare, disinstallare o unicamente scaricare i file senza poi proseguire al flash del firmware sulla Jetson. Altri parametri obbligatori sono `-product` dove occorre specificare quale prodotto si sta installando (nel nostro caso ovviamente Jetson) e `-version` dove va specificata la versione del firmware da installare. Noi abbiamo installato la versione più recente disponibile per la Jetson TX2, la 4.4. Occorre anche specificare il sistema operativo che verrà installato sulla Jetson tramite il parametro `-targetos Linux` e infine il modello stesso del modulo Jetson (trascritto sul fianco della scatola del Developer Kit). Nel nostro caso `-target P3310-1000`. Una volta lanciato il comando di installazione completo l'SDK Manager procederà all'installazione del sistema operativo e di tutti i driver, librerie e software necessari direttamente sulla Jetson. Conclusa l'installazione è possibile collegare il modulo

Jetson ad un monitor tramite cavo HDMI, una tastiera ed un mouse tramite HUB USB ed è possibile iniziare ad utilizzarla senza nessun'altra configurazione necessaria oltre alla creazione di un nome utente ed una password per il login. Il nome utente e la password che sono stati creati sono entrambi `jetsonunivpm`.

7.8.5 Installazione TensorFlow e Keras

Al primo avvio la Jetson presenta un sistema operativo Ubuntu con pre-installato Python 2.7 e svariate altre librerie per l'uso in ambito principalmente di Computer Vision come OpenCV, computazione gpu e deep learning. Per questo progetto è stato necessario installare svariate altre librerie, prime tra tutte TensorFlow e Keras. Innanzitutto per l'installazione di TensorFlow è stato necessario installare svariate librerie aggiuntive, nonché Python 3, come indicato in [38].

```
sudo apt-get update
```

```
sudo apt-get install libhdf5-serial-dev hdf5-tools libhdf5-dev zlib1g-dev
zip libjpeg8-dev liblapack-dev libblas-dev gfortran
```

Dopodichè è necessario installare e aggiornare pip3:

```
sudo apt-get install python3-pip
```

```
sudo pip3 install -U pip testresources setuptools
```

Ed infine alcune librerie da cui TensorFlow dipende per un corretto funzionamento come numpy, h5py, keras_preprocessing e keras_applications.

```
sudo pip3 install -U numpy==1.16.1 future==0.18.2 mock==3.0.5 h5py==2.10.0
keras_preprocessing==1.1.1 keras_applications==1.0.8 gast==0.2.2 futures
protobuf pybind11
```

Effettuate tutte le pre-installazioni necessarie si può procedere con l'installazione di TensorFlow. Per sapere quale versione è quella corretta si è utilizzata la tabella riportata in [39], e utilizzando poi il comando:

```
sudo pip3 install -extra-index-url https://developer.download.nvidia.com/compute/
redist/jp/v$JP_VERSION tensorflow==$TF_VERSION+nv$NV_VERSION
```

dove `$TF_VERSION` e `$NV_VERSION` sono rispettivamente la versione di TensorFlow e la relativa versione del container NVIDIA compatibile.

Per verificare che l'installazione sia stata eseguita correttamente, da un terminale si può eseguire il comando `python3` e successivamente `import tensorflow`. Se

TensorFlow è installato correttamente il comando verrà eseguito senza errori e in output verrà anche notificato se viene rilevata una GPU che verrà poi utilizzata per le istruzioni successive. Infine, per l'installazione di Keras basterà eseguire il relativo comando:

```
sudo -H pip3 install keras
```

7.8.6 Ambiente Jupyter e Colab

Una volta conclusa l'installazione di tutte le librerie necessarie, in teoria si è pronti a procedere con i test. Prima di farlo tuttavia abbiamo configurato la Jetson in modo da poter eseguire il codice attraverso dei notebook di Google Colab collegati alla Jetson attraverso una runtime locale. Per fare ciò è stato prima necessario installare l'ambiente Jupyter sulla Jetson. Di Jupyter e Google Colab si è già parlato abbondantemente in precedenza. Oltre all'installazione dell'ambiente Jupyter sulla Jetson è stato necessario installare il servizio di forwarding Jupyter attraverso la libreria:

```
pip install jupyter_http_over_ws
```

dopodichè occorre abilitare il servizio attraverso il comando:

```
jupyter serverextension enable -py jupyter_http_over_ws
```

ed infine avviare la sessione Jupyter:

```
jupyter notebook
```

```
-NotebookApp.allow_origin
```

```
= 'https://colab.research.google.com'
```

```
-port=8888
```

```
-NotebookApp.port_retries=0
```

Fatto questo otterremo un link che basterà inserire nell'interfaccia web del notebook Colab dopo aver selezionato l'opzione 'Connetti e runtime locale'.

7.8.7 Test effettuati

Sono stati effettuati diversi test per valutare le prestazioni in termini di memoria e in particolare il tempo richiesto dalla Jetson ad effettuare predizioni di un diverso numero di TAC variando parametri della rete come ad esempio il batch size utilizzato. Come già detto nell'introduzione, ci si è concentrati soltanto sul valutare la rete in

termini di tempi richiesti e non dell'effettiva correttezza delle predizioni, che non rientrano nelle specifiche di questo progetto. Si da quindi per assunto che la rete sia stata addestrata in precedenza al punto da raggiungere una accuratezza ritenuta soddisfacente. Per ogni test effettuato sono stati calcolati attraverso l'uso della libreria `time` di Python i tempi necessari ad eseguire le seguenti operazioni, calcolati in secondi.

- T_{tot} : il tempo impiegato in totale per la predizione di tutte le tac date in input.
- T_{tac} : il tempo medio impiegato per elaborare una tac.
- T_{slice} : il tempo medio impiegato per il singolo gruppo di 16 slice.

I test sono stati effettuati per gruppi di 5,10,15 e 20 tac, che si suppone dare in input alla rete tutte insieme ed estratte casualmente dal dataset a disposizione. Visto che ogni tac è formato da un numero variabile di slices, i risultati possono variare leggermente in esecuzioni successive. Lo stesso test con numero variabili di tac è stato ripetuto per batch size pari ad 1,2,5 e 10.

Di seguito i risultati per $n=5,10,15$ e 20 tac, con `batch_size=1`.

n	T_{Tot}	T_{tac}	T_{slice}
5	17.86	3.57	0.21
10	47.20	4.72	0.23
15	95.67	6.37	0.30
20	96.61	4.83	0.26

I risultati per $n=5,10,15$ e 20 tac, `batch_size=2`.

n	T_{Tot}	T_{tac}	T_{slice}
5	13.30	2.66	0.16
10	28.06	2.80	0.13
15	71.85	4.79	0.22
20	100.22	5.01	0.27

I risultati per $n=5,10,15$ e 20 tac, `batch_size=5`.

n	T _{Tot}	T _{tac}	T _{slice}
5	25.50	5.01	0.30
10	39.60	3.95	0.19
15	75.32	5.02	0.25
20	94.60	4.73	0.28

I risultati per n=5,10,15 e 20 tac, batch_size=10.

n	T _{Tot}	T _{tac}	T _{slice}
5	19.85	3.97	0.23
10	38.51	3.85	0.18
15	67.71	4.51	0.21
20	82.97	5.64	0.32

7.8.8 Risultati

Dai risultati possiamo vedere che i tempi complessivi scalano in modo piu' o meno lineare all'aumentare delle tac da elaborare. La media di tempo richiesto per la singola TAC e la singola slice tende a subire lievi fluttuazioni nei vari test eseguiti ma non variano in modo rilevante, mentre la maggior variazione nel tempo totale di esecuzione è spiegabile anche dal numero variabile di slices che possono avere le singole tac. Passando invece ad esaminare il batch size utilizzato possiamo vedere che aumentare il batch size non sempre risulta in un minor tempo richiesto, anche se questo accade comunque nella maggior parte dei casi. In particolare possiamo vedere che usare un batch size di 10 riduce in media di 10 secondi le predizioni relative a 15 tac, di ben 30 secondi quella di 15 tac e di quasi 20 secondi la predizione di 20 tac. Tuttavia aumentare il batch size richiede anche una maggiore allocazione di memoria: già con un batch size pari a 10 durante la predizione si sono verificati dei warning di poca memoria disponibile, ed aumentarlo ulteriormente potrebbe renderne difficile o direttamente impossibile l'uso.

Capitolo 8

Conclusioni e sviluppi futuri

In questa tesi abbiamo effettuato svariati esperimenti per verificare innanzitutto se era possibile per una rete imparare a differenziare tra le due tipologie di cancro al polmone, e poi altri esperimenti per verificare quale otteneva il miglior risultato. La prima ipotesi si può ritenere confermata: entrambe le reti hanno dimostrato la capacità di apprendere features discriminanti che permettono di distinguere tra le due tipologie. Questo risultato è incoraggiante e mostra che sono necessari ulteriori studi in modo da arrivare all'obiettivo finale che ci si era posti in questa tesi, ovvero quello di, idealmente, arrivare ad una rete con una accuratezza tale da permettere di ridurre i tempi di diagnosi e soprattutto evitare il doversi sottoporre ad una biopsia al paziente. Tra le due reti, la ConvLSTM sembra essere quella che ha ottenuto in generale i migliori risultati, permettendo inoltre di effettuare ulteriori esperimenti grazie alla possibilità, a differenza della C3D, di personalizzare la dimensione in input delle TC. Siamo andati ad esaminare come diverse tipologie di preprocessing delle immagini vanno ad influenzare le performance delle due reti: in particolare, abbiamo visto come non sempre un maggiore preprocessing corrisponde ad un miglioramento nelle performance della rete, è anzi possibile ritrovarsi con immagini a cui è stata rimossa fin troppa informazione, ed in generale il preprocessing effettuato e le dimensioni dell'immagine sembrano essere i due fattori che maggiormente influenzano le performance della rete. Spesso nei risultati ottenuti si è vista una tendenza di tutte le reti a predire maggiormente la classe adenocarcinoma a sfavore della classe tumore a cellule squamose, che si può spiegare con un leggero sbilanciamento del dataset a disposizione che conteneva più TC della prima tipologia. Per ovviare a questo problema, ovviamente, si avrebbe bisogno di più dati. Le TC utilizzate nel corso

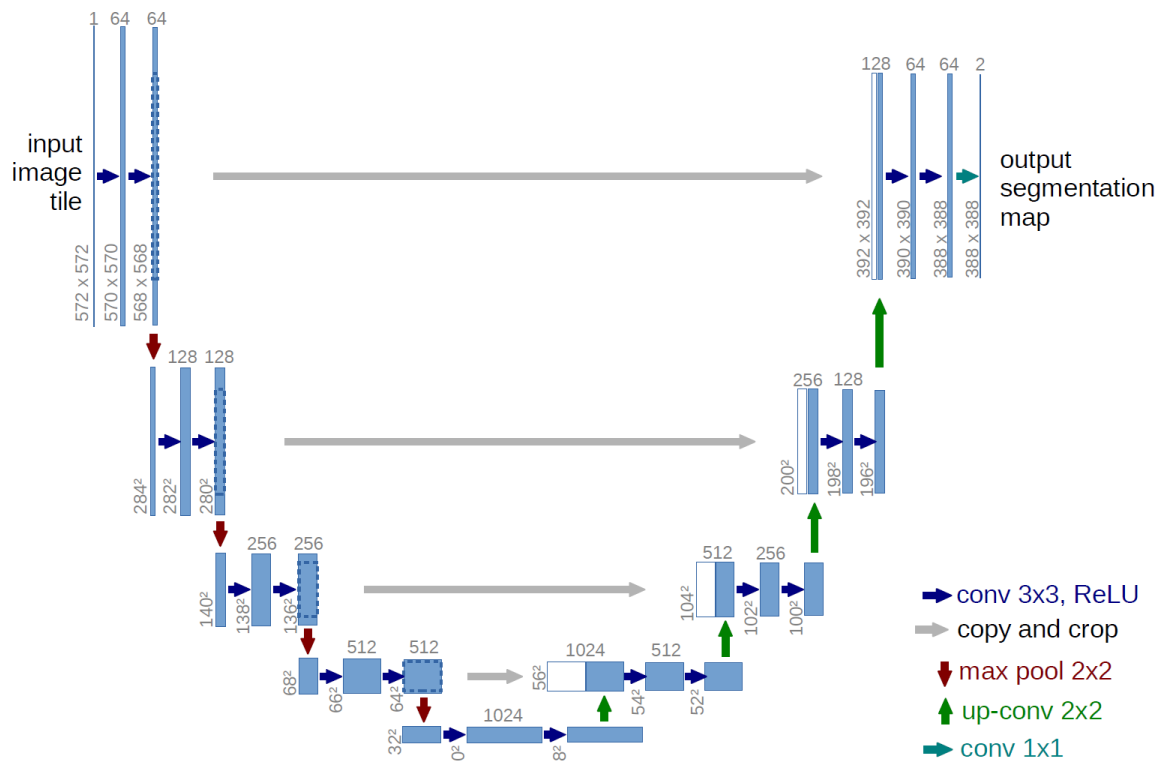


Figura 8.1: Architettura di una rete U-Net

di questa tesi sono state raccolte da una moltitudine di ospedali nel corso di molti anni. Altro possibile punto di sviluppo consisterebbe nel migliorare la segmentazione in generale, per cui sarebbe possibile provare ad utilizzare una rete convoluzionale appositamente addestrata ad effettuare una segmentazione migliore del polmone, come ad esempio la rete U-Net utilizzata da Ronneberger et al [40] (2015) oppure il successivo studio menzionato in precedenza di Kungpeng et al [9] (2019). Le reti U-Net hanno generalmente una struttura simile a quella presentata in 8.1. Una rete U-Net è suddivisa principalmente in due "cammini", il cammino di sotto-campionamento (downsampling) costituito come in una normale rete convoluzionale di un alternarsi di layer di convoluzione e maxpooling per ridurre le dimensioni dell'immagine, ed un cammino di espansione (upsampling), dove anzichè dare l'output dell'ultimo layer di convoluzione ad un layer di flatten e poi ad un classificatore si vanno ad alternare dei layer di up-convolutions e concatenazione, ottenendo in output, invece del risultato di un classificatore, una immagine segmentata ottenuta a partire dalle features estratte nel cammino di downsampling. E' possibile specificare le dimensioni desiderate dell'immagine di output, in modo da poterla rendere compatibile con l'input di

qualsiasi altra tipologia di rete si andrà ad utilizzare. Altro possibile sviluppo futuro interessante di questo studio è l'andare ad approfondire il rapporto tra performance di una rete e numero di filtri (e quindi numero di parametri e features che si vanno ad estrarre) utilizzati in combinazione con le dimensioni dell'immagine di input. Abbiamo visto che anche riducendo il numero di filtri utilizzati le performance di una rete possono addirittura andare ad aumentare se si danno in input immagini di dimensione e qualità maggiore, che è un altro passo avanti verso il nostro scopo di ottenere un procedimento end-to-end che, idealmente, permetterebbe di dare in input alla rete delle TC a risoluzione massima possibile con il minor livello di preprocessing necessario. Il numero di filtri migliore per un determinato problema di classificazione insieme ad altri parametri è un ambito che richiede sicuramente ulteriori studi: per problemi di classificazione di decine se non centinaia di classi, ovviamente il numero di features che la rete deve essere in grado di estrarre e apprendere deve essere molto alto, per permettere la distinzione tra le varie classi. In problemi di classificazione binaria invece potrebbe non essere necessario un numero così alto, avendo ottenuto performance incoraggianti anche con un numero di filtri relativamente molto basso, performance che potrebbero essere ulteriormente migliorate.

Bibliografia

- [1] J. Sidey-Gibbons. Machine learning in medicine: a practical introduction. 2019.
- [2] AIRC. Guida ai tumori: tumore del polmone, 2019.
- [3] ONA. Adenocarcinoma polmonare.
- [4] Pinuccia Valagussa Gianni Bonadonna, Gioacchino Robustelli Della Cuna. *Medicina oncologica*. Elsevier Masson, Milano, 2007.
- [5] X. Zhao. Agile convolutional neural network for pulmonary nodule classification using ct images. 2018.
- [6] Y. Wang. Novel convolutional neural network architecture for improved pulmonary nodule classification on computed tomography. 2020.
- [7] G. Perez. Automated lung cancer diagnosis using three-dimensional convolutional neural networks. 2020.
- [8] H. Shaziya. Comprehensive review of automatic lung segmentation techniques on pulmonary ct images. 2019.
- [9] Z. Kunpeng. Automatic lung field segmentation based on the u-net deep neural network. 2019.
- [10] S. E. Gerard. Multi-resolution convolutional neural networks for fully automated segmentation of acutely injured lungs in multiple species. 2020.
- [11] Q. Hu. An effective approach for ct lung segmentation using mask region-based convolutional neural networks. 2020.
- [12] T. J. B Lima. Lung ct screening with 3d convolutional neural network architectures. 2020.

Bibliografia

- [13] Schiaffonati Viola Somalvico Marco, Amigoni Francesco. *Intelligenza artificiale (Storia della scienza vol. IX)*. Istituto della Enciclopedia Italiana, Roma, 2003.
- [14] Future of Life Institute. Ai principles, <https://futureoflife.org/ai-principles/>.
- [15] Wikipedia. Apprendimento automatico, https://it.wikipedia.org/wiki/apprendimento_automatico.
- [16] Aditi Mittal. Understanding rnn and lstm, <https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e>.
- [17] Fawcett T. An introduction to roc analysis. 2005.
- [18] Narhede S. Understanding auc-roc curve, 2018.
- [19] Wikipedia. Tomografia computerizzata, https://it.wikipedia.org/wiki/tomografia_computerizzata/.
- [20] Wikipedia. Scala hounsfield, https://it.wikipedia.org/wiki/scala_hounsfield.
- [21] The Medical Imaging Technology Association. About dicom:overview, <https://www.dicomstandard.org/about>.
- [22] Project Jupyter. Jupyter project, <https://jupyter.org/>.
- [23] Google Colab. Google colab, <https://colab.research.google.com/notebooks/intro.ipynb>.
- [24] Python. Python, <https://www.python.org/>.
- [25] NumPy. Numppy,<https://numpy.org/>.
- [26] Keras. Keras, <https://keras.io/>.
- [27] TensorFlow. Tensorflow, <https://www.tensorflow.org/>.
- [28] Scikit-learn. Scikit-learn, <https://sklearn.org/>.
- [29] Scikit-image. Scikit-image, <https://scikit-image.org/>.
- [30] Pydicom. Pydicom, <https://pydicom.github.io/>.
- [31] Matplotlib. Matplotlib, <https://matplotlib.org/>.
- [32] Du T. Learning spatiotemporal features with 3d convolutional networks. 2015.
- [33] BVLC caffe. Bvlc caffe, <http://caffe.berkeleyvision.org/>.

- [34] NVIDIA. Nvidia, <https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-tx2/>.
- [35] NVIDIA. Nvidia, <https://developer.nvidia.com/embedded/jetson-tx2-developer-kit>.
- [36] NVIDIA. Nvidia, <https://developer.nvidia.com/nvidia-sdk-manager>.
- [37] NVIDIA. Nvidia, <https://docs.nvidia.com/sdk-manager/sdkm-command-line-install/index.html>.
- [38] NVIDIA. Nvidia, <https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html>.
- [39] NVIDIA. Nvidia, <https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform-release-notes/tf-jetson-rel.html#tf-jetson-rel>.
- [40] Brox T. Ronneberger O, Fischer P. U-net: Convolutional networks for biomedical image segmentation. 2015.