

Francesca Nocentini

Tesi di laurea

Tecniche di deep learning per la classificazione di immagini biomedicali

Relatore:

Prof. Gianluca Reali

Perugia, Anno Accademico 2020/2021
Università degli Studi di Perugia
Corso di laurea in Ingegneria Informatica ed Elettronica
Dipartimento di Ingegneria



0. Indice

1	Introduzione	4
2	Machine learning e reti neurali	6
2.1	L'importanza dell'apprendimento	6
2.2	Neuroni artificiali e deep learning	9
2.3	Addestramento di una rete	16
2.4	Overfitting e underfitting	20
2.5	Classificazione	22
3	Reti neurali convoluzionali	25
3.1	Funzionamento generale	25
3.2	Architettura generale di una CNN	26
3.2.1	Convoluzione	26
3.2.2	Padding	30
3.2.3	Funzioni di attivazione	31
3.2.4	Strati di subsampling: Pooling Layers	32
3.2.5	Fully connected layers	33
4	Ambiente di lavoro	35
4.1	Python	35
4.2	Tensorflow e Keras	36
4.3	Jupyter notebook e GoogleCollab	37
5	Implementazione delle reti e prove sperimentali	39
5.1	Obiettivo	39

5.2	CNN per la rilevazione della polmonite	40
5.2.1	Il dataset	40
5.2.2	Setup iniziale	41
5.2.3	Definizione degli iperparametri	42
5.2.4	Definizione e compilazione del modello	44
5.2.5	Creazione dei set di training e validation traminte l'uso dell'image flowing	47
5.2.6	Fase di fitting	50
5.2.7	Predizioni e grafici	52
5.2.8	Considerazioni finali	55
5.3	CNN per la classificazione di risonanze magnetiche cerebrali	57
5.3.1	Il dataset	57
5.3.2	Setup iniziale	59
5.3.3	Definizione degli iperparametri e dei modelli utilizzati	60
5.3.4	Creazione dei set di training e testing	68
5.3.5	Fase di fitting e predizioni	69
5.3.6	Considerazioni finali	76
6	Conclusioni	78
7	Codice	81
8	Bibliografia	89

1. Introduzione

Al giorno d'oggi, con l'avvento delle tecnologie di *imaging* biomedico, il numero di immagini che sono state catturate ed archiviate giorno dopo giorno negli ospedali e nei laboratori sta crescendo sempre di più. Questo però deve avvenire di pari passo all'avanzamento tecnologico, di modo che questo fornisca sistemi più robusti e all'avanguardia affinchè vengano raggiunti gli obiettivi di diagnosi e classificazione di vari tipi di patologie.

Sulla scia di ciò, per assistere medici e specialisti, le immagini iniziano a essere usate ed utilizzate per allenare sistemi intelligenti. Pertanto in virtù di questa grande quantità di immagini mediche (ecografie, mammografie, RM...), l'uso di metodi basate sulle *big data technologies*, come il *Machine Learning* (ML) e l'*Intelligenza Artificiale* è diventato fondamentale, anche e soprattutto come supporto all'equipe medica. Sono stati dunque proposti negli anni dei metodi per automatizzare il processo di analisi di immagine medica. Nel presente lavoro di tesi saranno analizzati ed utilizzati metodi di classificazione di immagini usando metodi di *deep learning* (DL), in particolare si fa uso di *reti neurali convoluzionali*. I metodi di DL non sono altro che un'evoluzione dei metodi di ML atti a trattare dati di più grande cardinalità e complessità come quelli inerenti al campo biomedico.

In particolare in una prima parte verrà fatta una discriminazione tra soggetti affetti da polmonite e soggetti sani tramite l'utilizzo di radiografie al petto; in una seconda parte invece il sistema viene allenato in modo tale da poter classificare immagini di risonanze magnetiche cerebrali tra 4 diverse diagnosi per il soggetto nell'immagine: glioma, meningioma, tumore ipofisario e soggetto sano. Prima di tutto è necessario precedere la

trattazione sulla realizzazione dei sistemi di classificazione con due sezioni introduttive su ciò che rappresentano il ML e il DL e su quale è il funzionamento in linea teorica generale di una rete neurale, per poter comprendere meglio quanto fatto nella fase sperimentale.

Più in dettaglio l'elaborato verrà suddiviso così:

- **Capitolo 2:** si introducono i concetti di base sull'apprendimento automatico, sul deep learning e le reti neurali e su ciò che significa addestrare una rete e farle acquisire capacità di *generalizzazione*. Si tratta di quelli che possono essere i principali problemi che si riscontrano nella realizzazione di sistemi di questo tipo e di cosa significa classificare.
- **Capitolo 3:** si analizza più nel dettaglio quella che è la teoria che definisce le strutture che sono state scelte nell'implementazione e che hanno reso possibile la classificazione d'immagine: le **CNN**.
- **Capitolo 4:** si introducono i linguaggi e le applicazioni utilizzate per elaborare i dati e per lavorare sull'apprendimento delle reti neurali, dunque si sottolinea quale è stato l'ambiente di lavoro.
- **Capitolo 5:** si descrivono le fasi per realizzazione del sistema di classificazione, a partire dal descrizione dei dataset fino ad arrivare alla visualizzazione delle previsioni sulle immagini stesse.
- **Capitolo 6:** si fa un breve resoconto dei risultati ottenuti e di un possibile sviluppo che può essere apportato sui sistemi realizzati.
- **Capitolo 7:** si riportano due esempi di codice utilizzati per lavorare sulle immagini e classificarle.

2. Machine learning e reti neurali

2.1 L'importanza dell'apprendimento

Le reti neurali [1] sono alla base del deep learning, che è un sottocampo del machine learning. Questa ultima branca è fondamentale nello studio e nello sviluppo delle Intelligenze Artificiali, sistemi basati sull'apprendimento automatico partendo dallo studiare i dati in input per fornire dati più vicini possibile a quelli desiderati. Sostanzialmente un algoritmo di machine learning è un algoritmo capace di apprendere dai dati.

"Si dice che un programma apprende dall'esperienza E con riferimento a alcune classi di compiti T e con misurazione della performance P, se le sue performance nel compito T, come misurato da P, migliorano con l'esperienza E." [2]

Tom M. Mitchell¹

Il compito principale del machine learning è che una macchina sia in grado di generalizzare dalla propria esperienza. Per generalizzazione si intende l'abilità di una macchina di portare a termine in maniera accurata esempi o compiti nuovi, che non ha mai affrontato, dopo aver fatto esperienza su un insieme di dati di apprendimento.

La macchina ha il compito di costruire un modello probabilistico generale dello spazio delle occorrenze di un determinato fenomeno da apprendere tramite dei *training examples*,

¹Tom Michael Mitchell è un computer scientist americano e professore universitario. È conosciuto per aver contribuito attivamente allo sviluppo degli studi sul machine learning e le intelligenze artificiali.

in maniera tale da essere in grado di produrre previsioni sufficientemente accurate quando sottoposta a nuovi casi. Proprio a questo proposito, al fine di valutare la bontà di un algoritmo di machine learning, dobbiamo stimare un andamento qualitativo della sua performance P, la quale molto spesso è specifica per un determinato compito T che il sistema deve compiere.

I compiti dell'apprendimento automatico vengono tipicamente classificati in tre ampie categorie, sulla base dell'esperienza E a cui sono sottoposti nel processo di apprendimento, dunque a seconda della natura del "segnaletico" utilizzato per l'apprendimento o del "feedback" disponibile al sistema di apprendimento. Queste categorie, anche dette paradigmi, sono [3]:

- **apprendimento supervisionato**, in cui al modello vengono forniti degli esempi nella forma di possibili input e output desiderati e l'obiettivo è quello di estrarre una regola generale che associa l'input all'output corretto;
- **apprendimento non supervisionato**, in cui il modello ha lo scopo di trovare una struttura negli input forniti, senza che questi vengano etichettati in alcun modo;
- **apprendimento per rinforzo**, il quale punta a realizzare agenti autonomi in grado di scegliere azioni da compiere per il conseguimento di determinati obiettivi tramite interazione con l'ambiente in cui sono immersi.

Il tipo di apprendimento di interesse per il lavoro di tesi è quello supervisionato, in quanto si utilizzano coppie input-output rappresentate da immagini e label che le identificano.

Tra i compiti più importanti dell'apprendimento supervisionato vi sono la classificazione e la regressione, i quali si distinguono a seconda di come viene considerato l'output del sistema di apprendimento. Nella prima gli output sono divisi in due o più classi e il sistema di apprendimento deve produrre un modello che assegna gli input non ancora visti a una o più di queste [4]. Si dice che i valori assunti dall'output sono *qualitativi*. Nella seconda invece si può dire che a differenza della prima i dati in output sono continui, cioè non riguardano una categorizzazione: si dice che i valori assunti dall'output sono *quantitativi*, in quanto restituiscono la stima di una misura.

La classificazione è il compito su cui verte il presente lavoro di tesi. Per tale scopo, la performance P dell'algoritmo di machine learning si valuta misurando l'accuratezza del modello cioè la percentuale di esempi per i quali il modello elabora un output corretto sul totale delle osservazioni. Un altro parametro di riferimento può essere il rate di errore,

SEZIONE 2.1. L'IMPORTANZA DELL'APPRENDIMENTO

definito invece come la proporzione di esempi per cui il sistema elabora un output sbagliato.

È importante verificare come l'algoritmo riesca a valutare dati che non abbia mai visto. Misure di performance vengono effettuate usando un insieme di dati chiamato insieme di *test*. Questo insieme di dati è di solito diverso dall'insieme di esempi usati per allenare il sistema (insieme di *addestramento* o *training*) affinchè si possano fare valutazioni più precise ed indipendenti da quanto sia stato efficace l'addestramento. Di solito l'esperienza E corrisponde ad un intero *dataset*, una collezione di dati (che possono essere sotto forma di vettori, immagini, suoni ecc.) suddiviso in classi tale per cui ad ogni elemento è associata una ed una sola di queste. Tali dati, essendo utilizzati in fase di addestramento, è estremamente importante che siano costruiti in maniera corretta. Dunque il punto di partenza per un buon addestramento è sicuramente un buon dataset.

Come accennato sopra, l'apprendimento supervisionato è una forma di apprendimento automatico in cui al sistema da allenare si sottopone in input un insieme di addestramento formato da esempi etichettati con il rispettivo valore di output desiderato, e tra gli input e output si vuol trovare una corrispondenza. Questo processo fa tipicamente riferimento a tecniche di tipo statistico.

L'apprendimento consiste nell'osservazione di una quantità di esempi di un vettore casuale \mathbf{x} a cui è associato un vettore \mathbf{y} per poi apprendere come predire \mathbf{y} da \mathbf{x} [5], stimando una distribuzione di probabilità condizionata² $p(\mathbf{y}|\mathbf{x})$ per l'output, che consenta di ottenere anche buoni valori di confidenza per l'algoritmo.

I valori di \mathbf{y} fungono da guida per indirizzare e migliorare l'apprendimento di contro al caso non supervisionato in cui l'algoritmo deve trattare i dati senza l'ausilio di questa guida.

Un esempio di apprendimento supervisionato è il training di un sistema col compito di riconoscimento delle immagini. In questo caso dobbiamo specificare quale oggetto appare in ogni foto. Possiamo fare questo con un codice numerico che caratterizzi le varie classi, per esempio usando 0 per le persone, 1 per le macchine, 2 per i gatti, e così via, a seconda di quello che è l'elemento di interesse da riconoscere.

²Date due variabili aleatorie X e Y , la distribuzione condizionata di Y dato X è la probabilità di Y quando è conosciuto il valore assunto da X .

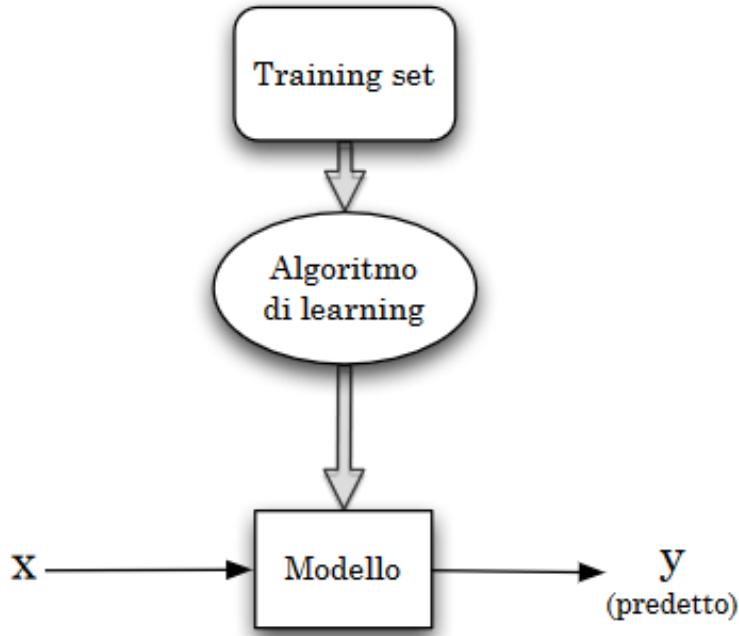


Figura 2.1: Schema dell'apprendimento supervisionato [6].

2.2 Neuroni artificiali e deep learning

Le *reti neurali* [7] si riferiscono storicamente alla rete di neuroni che si trovano nel cervello dei mammiferi, e la loro struttura ne ha ispirato gli algoritmi. I neuroni sono le unità fondamentali di calcolo, e sono connessi tra loro in reti per processare dati. Essi rispondono a stimoli esterni, così come le reti neurali prendono dati in input, si allenano nel riconoscere dei pattern ripetuti all'interno dei dati, e sono in grado di predire l'output per un set di dati simili.

La corteccia cerebrale umana contiene circa 10^{10} neuroni. Sono collegati tra loro da filamenti nervosi (assoni) che si diramano e terminano in sinapsi, connessioni verso altri neuroni. Le sinapsi connettono verso i dendriti, diramazioni che si estendono dal corpo della cellula neurale e sono atti a ricevere impulsi da altri neuroni nella forma di segnali elettrici. Nelle reti neurali biologiche lo spessore dei dendriti definisce il peso ad esso associato. La rete risultante di neuroni tra loro connessi nella corteccia cerebrale è responsabile del processare immagini, audio e dati sensoriali. In presenza di una

differenza di potenziale fra esterno e l'interno della cellula, il neurone si attiva trasmettendo un impulso elettrico attraverso il suo assone che provoca la liberazione di un neurotrasmettore.

Nelle reti neurali artificiali, il modo in cui le informazioni sono processate e i segnali sono trasferiti sono ampiamente idealizzati da tali concetti.

Neuron

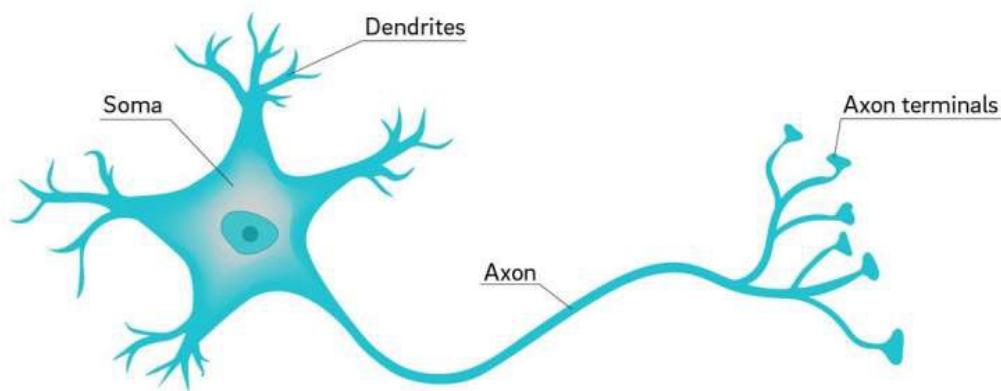


Figura 2.2: Un semplice schema di neurone. [8]

Il primo modello computazionale di neurone è stato proposto nel 1943 da Warren McCulloch e Walter Pitts³. McCulloch and Pitts hanno pensato ad un modello di neurone come un'unità di *threshold* (soglia) binaria.

Essa possiede due possibili stati: attivo o inattivo.

Per ottenere il segnale di output il neurone elabora una somma pesata degli input: se la somma eccede un certo valore di threshold, lo stato del neurone risulta attivo, altrimenti inattivo. Il modello, in intervalli discreti di tempo $t = 0, 1, 2, 3\dots$, svolge del calcolo

³McCulloch e Pitts sono stati un neurofisiologo e un logico americani famosi per essere stati i primi a realizzare il primo modello matematico di rete neurale

computazionale.

Lo stato del neurone numero j al passo t è identificato come:

$$s_j(t) = \begin{cases} 1 & \text{attivo} \\ -1 & \text{non attivo} \end{cases}$$

Ricevuti in ingresso N stati $s_j(t)$, il neurone numero i calcola

$$s_i(t+1) = \operatorname{sgn}\left(\sum_{j=1}^N w_{ij}s_j(t) - \theta_i\right) = \operatorname{sgn}(b_i(t)) \quad (2.1)$$

dove w_{ij} è il peso associato agli stati s_1, \dots, s_N , il cui primo indice i si riferisce al neurone che fa il calcolo, invece l'indice j rappresenta tutti i neuroni connessi al neurone i (figura 2.3). Il valore del peso è rappresentato da un numero reale, che riproduce lo spessore e la conducibilità delle sinapsi neurali. Il segnale di uscita al passo $t + 1$ è ricavato attraverso la somma pesata degli ingressi, detta livello di attivazione o *local field* $b_i(t)$. Alla somma pesata vediamo nella (2.1) essere sottratto un valore di *threshold*, denotato come θ_i .

Nel campo del deep learning di cui si parlerà a breve esso è conosciuto anche come *bias*, ed è necessario per la robustezza della rete neurale. Può essere identificato come un ulteriore canale di ingresso s_0 che serve per alzare o abbassare la soglia di attivazione del neurone (*threshold*), dove per soglia indichiamo il livello di tensione oltre al quale il neurone risulta attivo.

La somma pesata e shiftata viene poi passata per una determinata funzione f , che nel caso

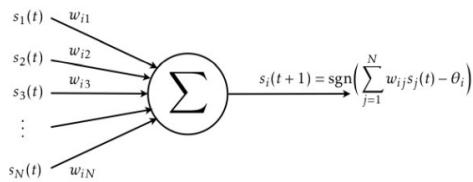


Figura 2.3: Schema di un percettrone. [7]

del neurone di McCulloch e Pitts è la funzione segno $\operatorname{sgn}()$.

Nel deep learning la funzione segno è molto spesso sostituita da una funzione f chiamata *funzione di attivazione*. La funzione di attivazione definisce l'uscita di un neurone in

funzione del suo livello di attivazione $b_i(t)$. Una scelta comunemente utilizzata è la *sigmoide*:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Un'altra molto importante e anch'essa utilizzata è la *ReLU* (Rectified Linear Unit):

$$f(x) = x^+ = \max(0, x)$$

Tale funzione è una di quelle maggiormente impiegate negli algoritmi di deep learning per la rapidità di apprendimento che permette di sviluppare nelle reti formate da vari strati di neuroni e per altri fattori che saranno approfonditi in seguito.

La formula (2.1) è anche la formula che definisce la *regressione logistica*, l'algoritmo d'apprendimento base di cui si compongono le reti neurali. La computazione fatta nel singolo neurone viene poi eseguita in maniera parallela per tutti i neuroni e gli output di s_i diventano gli input per i neuroni dello strato successivo a quello i -esimo. Lo scopo quindi di McCulloughs è stato quello di costruire un modello che simulasse le dinamiche delle reti neurali reali.

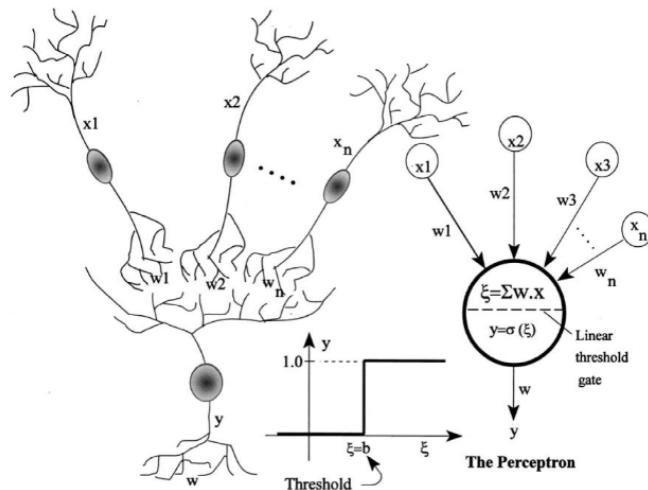


Figura 2.4: Figura che rappresenta come le reti neurali siano effettivamente ispirate alle reti neurali biologiche. A sinistra vi è una rete di neuroni reali e a destra un modello di percettrone.

Dato che una rete neurale efficiente deve essere in grado di apprendere e fornire uscite

a fronte di input anche sconosciuti, occorre allenare la rete con degli algoritmi. L'addestramento della rete neurale si basa sulla presentazione al suo ingresso di una serie di esempi, la cui natura la rete è in grado di riconoscere, che costituiscono il dataset di training. Le risposte che la rete fornisce vengono confrontate con quelle attese e si calcola l'errore tra le due. Sulla base di tale errore i pesi vengono modificati istante per istante, e la procedura viene ripetuta fino a che la rete non fornisce un errore che sia al di sotto di una determinata soglia prestabilita.

Una rete neurale si dice stratificata quando è formata da vari strati di neuroni, tali che ognuno di loro sia connesso con quelli dello strato successivo, ma senza che ci siano connessioni tra neuroni del medesimo strato. Il neurone pensato da McCullough è chiamato anche Percettrone e possiede solamente uno strato di input e uno di output. Di solito si lavora con reti in cui vi sono uno o più *hidden layers*, i quali non comunicano direttamente con l'esterno. Queste reti sono chiamate MPL (Multi-Layer-Perceptron).

Il primo vero miglioramento rispetto al Perceptron è stato l'*Adaline* (ADaptive LInear NEuron), in quanto utilizza proprio una funzione di attivazione lineare per regolare il vettore dei pesi al posto della funzione a gradino.

Una rete neurale artificiale in cui per ogni strato il segnale di ingresso viaggia sempre in avanti dall'ingresso all'uscita, senza creazione di cicli, si chiama *feedforward*.

L'apprendimento profondo [9], in inglese Deep Learning (DL) è quel campo di ricerca del ML e dell'Intelligenza Artificiale che studia le tecniche e i sistemi per consentire alle macchine di apprendere in autonomia in modo profondo. Nello specifico, questa branca viene considerata una categoria sottostante al machine learning, in quanto si tratta di un approccio per l'apprendimento automatico dei sistemi informatici.

In altre parole, per apprendimento profondo si intende un insieme di tecniche basate su reti neurali artificiali organizzate in diversi strati, dove ogni strato calcola i valori per quello successivo affinché l'informazione venga elaborata in maniera sempre più completa con l'aumentare della profondità. Lo sviluppo del deep learning è motivato in parte dal fallimento degli algoritmi tradizionali di ML nel tentare di svolgere compiti delle AI, come ad esempio speech recognition e object recognition. Il principio di funzionamento è lo stesso delle reti neurali classiche, con una differenza che risiede nel numero molto elevato di livelli nascosti di neuroni intermedi. Altre sfide che hanno portato allo sviluppo

dell'apprendimento approfondito è la difficoltà degli algoritmi classici nel generalizzare su problemi con dati multi-dimensional. Il fenomeno legato alla grande dimensionalità dei dati è conosciuto come *curse of dimensionality* [10]: questo fenomeno si presenta spesso in molti campi della computer science, in particolare nel machine learning. È stato introdotto questo problema per la prima volta dal matematico Richard Bellman e indica che il numero di campioni necessari per stimare una funzione arbitraria con un dato livello di accuratezza cresce esponenzialmente rispetto al numero di variabili di input (cioè dimensionalità) della funzione.

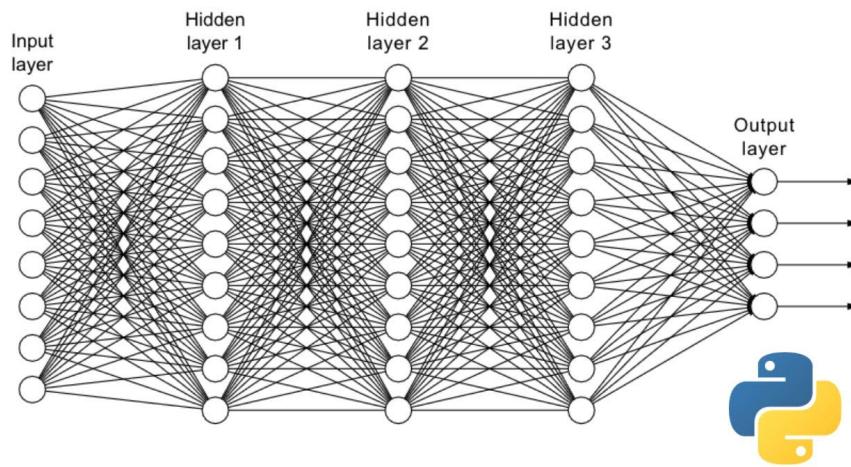


Figura 2.5: Esempio di rete neurale profonda con 3 strati nascosti. [11]

All'aumentare del numero dei dati aumenta anche il numero delle *features* (cioè il numero di parametri con cui i dati possono essere descritti), ma all'aumentare delle features diminuisce anche la capacità predittiva del sistema con un conseguente peggioramento delle prestazioni. Come già detto nel precedente capitolo, con i metodi convenzionali di machine learning bisogna costruire gli estrattori delle feature dei dati di input. L'accuracy e la precisione delle predizioni sono fortemente influenzate dall'abilità di colui che realizza il modulo di estrazione, che richiede inoltre un forte sforzo ingegneristico.

Un vantaggio fondamentale delle reti di deep learning è la possibilità di migliorare le prestazioni con l'aumentare del formato dei dati.

Un'organizzazione gerarchica dei dati permette di condividere e riutilizzare informazioni estratte durante l'elaborazione e di selezionare e scartare dettagli inutili

lungo una gerarchia. Rispetto ad una architettura semplice a tre strati (input-strato con unità nascoste-output), una architettura multi-strato permette di distribuire meglio un grande numero di nodi su più livelli riducendo il costo computazionale elevato che si avrebbe se i nodi fossero tutti localizzati su uno solo attenuando l'ingente utilizzo di memoria che comporterebbe una struttura meno profonda.

In generale le tecniche di deep learning stanno diventando sempre più utili anche in quello che è il campo della *computer vision*, cioè l'area di studio delle AI che si occupa della ricerca sull'automatizzazione dell'informazione visuale, e in particolare anche dell'analisi di dati biomedici, come nel caso del presente lavoro di tesi.

Le Deep Feedforward Networks [12], chiamate anche multilayer perceptrons (MLPs), sono degli strumenti molto importanti usati nel DL. L'obiettivo delle MLPs è approssimare una funzione f^* . Ad esempio, se si vuol costruire un classificatore si esegue una mappatura dell'input x su una categoria y : $y = f^*(x)$. Una rete di tipo feedforward definisce una mappatura $y = f(x, \theta)$ e cerca il parametro θ che renda migliore l'approssimazione della funzione. Questi modelli sono chiamati feedforward perché l'informazione fluisce prima attraverso la funzione che deve valutare x , poi lungo le elaborazioni intermedie usate per definire f fino all'output y . Questo tipo di reti sono di solito rappresentate dalla composizione di più funzioni ad esempio f^1, f^2 e f^3 connesse da una catena:

$$f(x) = f^3(f^2(f^1(x)))$$

In questo caso f^1 sarà chiamato primo strato (first layer), f^2 secondo strato e così via. La lunghezza della catena indica la *profondità* del modello. Lo strato finale della rete è chiamato strato di output.

Durante l'addestramento della rete si cerca di approssimare $f(x)$ ad una funzione $f^*(x)$ valutata su differenti esempi di addestramento.

Questi specificano direttamente cosa deve fare lo strato di output ad ogni punto x : deve produrre un valore il più vicino possibile alla label di target y .

Il comportamento degli altri strati non è direttamente specificato dai dati di addestramento; l'algoritmo di apprendimento deve decidere come usarli per produrre l'output desiderato per implementare al meglio una approssimazione di f^* . Siccome i dati di addestramento non mostrano l'output desiderato per ognuno di questi strati vengono

chiamati *strati nascosti*. La dimensione degli strati nascosti determina la larghezza del modello.

Lo strato rappresenta una funzione a valori vettoriali formata da molte *unità* che agiscono in parallelo, ognuna rappresentante una funzione a valori vettoriali che restituisce un valore scalare. Ogni unità è assimilabile ad un neurone nel senso che essa riceve l'input da altre unità ed elabora un output che verrà mandato a sua volta come input ad altre unità. Il valore di output viene chiamato valore di attivazione, che è analogo a quello delle reti neurali non profonde.

2.3 Addestramento di una rete

Come è stato già detto in precedenza, il punto di partenza per l'addestramento in generale, ma ancora di più per addestrare una rete neurale, è avere a disposizione un dataset formato da un insieme di training che verrà somministrato in input alla rete per estrarne automaticamente le features, e un insieme di test con dati completamente nuovi al sistema per verificare l'efficacia dell'addestramento stesso.

Per semplicità, assumiamo che il problema che vogliamo risolvere sia un problema di classificazione binaria. Per esempio vogliamo decidere se, data una certa immagine, questa rappresenti o meno un gatto.

Rappresentiamo la nostra immagine come un vettore x unidimensionale di byte. Fissiamo $y = 1$ nel caso in cui l'immagine rappresenti un gatto, $y = 0$ nel caso in cui nell'immagine non ci sia un gatto. La nostra unità di regressione logistica dovrà dunque calcolare un'approssimazione di y ; più precisamente calcolerà le probabilità che nell'immagine ci sia un gatto. Dato x , vogliamo dunque calcolare l'approssimazione $\hat{y} = P(y = 1|x)$. Perché ciò avvenga, serve che i parametri w e θ siano opportunamente configurati, ed è qui che entra in gioco il concetto di apprendimento. Durante l'addestramento, dopo aver fornito l'input, la rete produce un output in forma di vettore di risultati. Bisogna calcolare una funzione che misuri l'errore (o la distanza) tra i risultati di output e i risultati desiderati, cioè i valori con cui vengono etichettati i dati nell'apprendimento supervisionato.

La funzione che ci permette di farlo è chiamata *funzione di errore* o di perdita. Una delle funzioni di errore utilizzata nell'apprendimento supervisionato è la MSE (Mean Squared

Error):

$$MSE = \frac{1}{2} \sum (out - target)^2 \quad (2.2)$$

Nel deep learning è molto utilizzata *funzione di entropia incrociata*, funzione di perdita che serve per quantificare la differenza tra due distribuzioni di probabilità: quella che vorremmo il nostro sistema raggiungesse, e quella che il nostro sistema ha elaborato statisticamente fino a quel momento. Viene descritta dalla seguente formula:

$$H(h, q) = - \sum_x p(x) \log(q(x))$$

dove p è la funzione di distribuzione da raggiungere su un insieme di eventi x , mentre q è quella fino ad ora elaborata. Ogni volta questa funzione ci indica quanto la nostra stima fatta si discosti dalla realtà.

La rete in fase di addestramento modifica i suoi parametri interni variabili in modo da minimizzare la funzione di costo. Questi parametri sono chiamati pesi e spesso sono numeri reali che possono essere visti come delle "manopole" che definiscono la funzione di input-output della rete. Per modificare in maniera appropriata il vettore dei pesi, l'algoritmo di apprendimento calcola un vettore gradiente che, per ogni peso, indica di quanto l'errore cresce o decresce se i pesi vengono incrementati di una quantità infinitesima. Il vettore gradiente, calcolato sulla funzione di costo indica la direzione più ripida e veloce su come andare a aggiornare i pesi stessi, portandosi sempre più vicino a dove l'errore di stima dell'output è minore in media.

L'approccio più utilizzato al fine di minimizzare l'errore è la procedura chiamata *gradiente stocastico decrescente* [1]. Esso consiste nel mandare in input alla rete alcuni esempi, calcolandone l'output con la (2.1), l'errore associato (ad esempio con la (2.1)) e il gradiente medio di questa ultima variando i pesi in maniera infinitesima. La procedura è iterata per molti set piccoli di esempi presi dall'insieme di addestramento finché la media della funzione oggetto non smette di decrescere. Si parla di gradiente stocastico perché ogni piccolo insieme di esempi fornisce una stima del gradiente medio su tutti gli altri.

A titolo di esempio, si mostra successivamente come funziona l'algoritmo del gradiente discendente per un semplice percepitrone.

1. La rete prende in ingresso dei valori randomici iniziali per i pesi W e un valore

costante, il learning rate η . Tale valore determina lo step da fare ad ogni iterazione nella direzione indicata dal gradiente.

2. Si calcola la funzione di costo con i parametri attuali;
3. Si aggiornano i valori dei pesi usando la funzione di aggiornamento dei pesi W che è:

$$W^{(\rho)} = W^{(\rho-1)} - \eta \frac{\partial E}{\partial W} \quad (2.3)$$

dove ρ indica il valore dei pesi al passo precedente e η il learning rate.

4. Alla fine l'algoritmo ritorna il costo minimizzato per la funzione di errore.

Occorre dunque vedere come il valore dei pesi ha effetto sulla funzione di errore. Si applica la regola della catena di derivazione:

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial \text{out}} \frac{\partial \text{out}}{\partial \text{in}} \frac{\partial \text{in}}{\partial W}$$

dove in è la funzione che è combinazione lineare di input e pesi, mentre out è l'output che si ottiene facendo passare in per la funzione di attivazione, per esempio la sigmoide. Nel caso in cui si abbia una rete senza hidden layers la regola della catena si applica nel modo seguente, se vi fossero più strati però la procedura è analoga andando ad applicare tale regola per ognuno degli strati. Se si vanno a valutare le nostre derivate per il caso del percettrone, si può facilmente dimostrare che:

$$\begin{aligned} \frac{\partial E}{\partial \text{out}} &= (\text{out} - \text{target}), \\ \frac{\partial \text{out}}{\partial \text{in}} &= \text{out}(1 - \text{out}), \\ \frac{\partial \text{in}}{\partial W} &= \text{in} \end{aligned}$$

Si moltiplicano tra loro dunque tutti i termini derivativi e poi si applicano all'equazione principale (2.3) e si aggiornano i valori dei pesi di conseguenza, fintanto che la media dell'errore non converge, ovvero smette di decrescere.

Tale algoritmo nelle prove sperimentali ha effettiva realizzazione in quello che è chiamato *l'ottimizzatore* (uno dei più efficienti è proprio SGD (Stochastic Gradient Descent), ma vi sono anche Adam, ADALine...). Tutti però seguono l'idea di base del gradiente discendente. Infine terminata questa procedura si passa alle valutazioni sulla performance della rete; essa viene misurata con gli esempi dell'insieme di test.

Il metodo migliore per minimizzare la funzione di errore è proprio quello di usare il gradiente. Nelle reti MLP (Multi-Layer-Perceptron) si effettuano una serie di derivate successive e ogni volta si trova il punto che annulla la derivata della funzione di errore. Questo procedimento è piuttosto semplice nelle reti neurali più basilari, mentre inizia ad essere più complicato nelle reti neurali con più strati perché vi possono essere più minimi locali. In questo caso si gioca sul valore del learning rate. Un valore del learning rate (di solito compreso tra 0 e 1) troppo alto farà fare un salto troppo grande verso il minimo della funzione di costo, facendo perdere alla rete capacità di convergenza (rischiando in alcuni casi anche la divergenza) mentre un learning rate troppo basso rallenta notevolmente il training e può comportare il fatto che l'algoritmo rimanga fermo nel calcolo di un minimo locale indesiderato per la funzione di errore.

Dunque, il ciclo di apprendimento di una rete neurale feedforward può essere sintetizzato in due fasi:

- fase di FEEDFORWARD: si fanno passare i pattern dei dati di addestramento (input features) attraverso la rete per ottenere un output;
- fase di BACKPROPAGATION: si calcola la differenza tra l'output e il valore di target (di base si calcola l'errore di predizione) e si utilizza così l'algoritmo Gradient Descent per aggiornare i valori dei pesi. In pratica si riporta il valore dell'errore all'ingresso, ne si calcola la sua derivata rispetto ad ogni peso della rete e ogni volta si aggiornano i valori dei pesi del modello.

Dopo varie iterazioni dei due passi si ottiene una maggiore accuratezza di predizione per la rete.

Nelle reti neurali profonde si utilizzano gli algoritmi di back-propagation per facilitare il procedimento di calcolo del gradiente. In questi algoritmi i pesi sono calcolati partendo dall'ultimo strato di neuroni procedendo all'indietro. Un ciclo di presentazione alla rete di tutti gli eventi appartenenti all'insieme di addestramento è detto *epoca*.

La procedura di propagazione all'indietro calcola il gradiente di una funzione oggetto in funzione dei pesi associati ai propri parametri proprio applicando semplicemente la regola di derivazione della catena: la derivata, o proprio il gradiente, di una funzione rispetto ai valori di input può essere calcolata propagando all'indietro, dall'output del modello fino ad arrivare al primo strato. Partendo dall'output, cioè lo strato dove la rete produce le sue predizioni, il gradiente fluisce attraverso tutto il modello fino ad arrivare all'input, dove sono forniti i dati su cui sono fatte le predizioni.

Una volta calcolati questi gradienti è facile trovarne l'espressione dipendente dai pesi associati ai parametri della funzione oggetto.

2.4 Overfitting e underfitting

L'abilità di un algoritmo di performare bene su input mai osservati precedentemente si chiama generalizzazione. Un buon algoritmo di machine learning è un algoritmo che possiede errore di generalizzazione minimo.

I termini *overfitting* e *underfitting* sono nati proprio come risposta alle defezienze di cui il modello può soffrire. Pertanto sapere quanto il modello commette errori, significa sapere quanto questo va in overfitting o underfitting. Come è possibile influenzare l'errore sull'insieme di test quando questo insieme non si conosce a priori?

Il campo della teoria dell'apprendimento statistico fornisce alcune risposte. Attraverso delle assunzioni riguardanti la maniera in cui sono stati raccolti i dati si possono studiare matematicamente le relazioni che intercorrono tra l'errore dell'insieme di addestramento e l'errore sull'insieme del test.

Quindi, dopo aver generato l'insieme di addestramento, si scelgono e si variano i parametri per ridurre l'errore sull'insieme, che dovrebbe corrispondere anche all'errore sull'insieme di test. I fattori che determinano quanto sia efficiente un algoritmo di apprendimento sono:

1. la sua abilità nel minimizzare l'errore di addestramento;
2. la sua abilità nel minimizzare il gap tra errore di addestramento e l'errore di test.

L'**underfitting** è quando il modello non riesce a rendere l'errore di addestramento sufficientemente piccolo. Questo significa che il modello è troppo semplice e non si riescono a identificare un numero sufficiente di features, il che rende impossibile al modello di apprendere da quel dataset. In termini di machine learning significa che c'è stato un focus troppo basso sul set di training, dunque il modello non è nemmeno in grado di testare correttamente. L'**overfitting** è il caso in cui il modello ha imparato molto bene dal suo dataset di training, ma non è in grado di generalizzare bene. Questo significa che c'è un forte gap tra errore di addestramento ed errore di test. In termini di machine learning vuol dire che che c'è stato un eccessivo focus nel training set, e che le relazioni stabilite tra i neuroni non sono valide per valutare correttamente nuovi dati. Generalmente gli algoritmi di machine learning performano meglio quando la capacità dei dati è appropriata per la reale complessità del compito che devono svolgere e la quantità di dati dell'insieme di addestramento forniti. Modelli con capacità insufficiente non sono capaci di svolgere compiti complessi. Modelli con grande capacità possono risolvere compiti difficili, ma nel caso in cui la capacità sia più grande di quella richiesta del compito da svolgere potrebbero andare in overfitting. In effetti nel machine learning i modelli sono soggetti al problema del *trade-off varianza-bias*. Questo nasce come conflitto nel provare a minimizzare simultaneamente le due fonti di errore che impediscono al modello di generalizzare:

- errore di bias, che è consequenziale ad errate assunzioni fatte dall'algoritmo di learning, per rendere il dataset più facilmente allenabile (underfitting).
- errore di varianza, dato dalla forte sensitività del modello a fronte di piccoli cambiamenti del training set, con conseguente aumento di rumore (overfitting).

È impossibile minimizzare entrambe, ma è possibile raggiungere un buon compromesso tra le due.

Un modo per evitare underfitting (bias alto) è quello di utilizzare una maggior quantità di dati ma questo non sempre funziona. Altrimenti si può incrementare la complessità del

modello, riducendo il rumore dei dati oppure aumentando il numero di epochs e quindi incrementando la durata del training.

Tutto ciò però nella misura tale per cui non si ricada nell'overfitting (alta varianza), che può essere evitato ad esempio riducendo la complessità del modello, in quanto dati troppo ‘rumorosi’ possono portare a valutazioni erronee e poco generalizzabili. Un altro modo è quello di ridurre il numero di parametri o come si vedrà in seguito nelle CNN, usando dei Dropout. Un'altra tecnica è quella di utilizzare alcune funzioni, come EarlyStopping in Keras che valutano passo passo l'andamento del training e lo fermano non appena si nota che il modello non è più in grado di generalizzare, cioè prima che cada in overfitting.

Si andrà più nel dettaglio di Keras e delle sue funzioni in seguito (sez. 4).

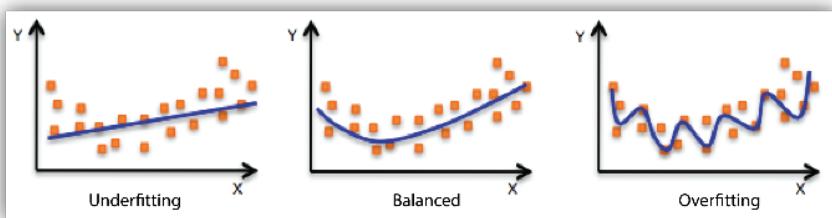


Figura 2.6: I grafici mostrano cosa significa underfitting e overfitting in maniera visiva. [13] Si può vedere come la prima curva di regressione non vada a suddividere in maniera precisa i dati in rosso, mentre l'ultima curva separi bene tutti i dati classificandoli, ma è una curva troppo precisa e specifica per i dati di interesse e che probabilmente non andrà a dare risultati corretti se i dati in ingresso si discostano anche leggermente da quelli rossi in figura. Al centro invece la curva risulta essere bilanciata.

2.5 Classificazione

Come detto in precedenza, in questo lavoro di tesi le reti saranno addestrate al fine della classificazione [14] delle immagini di raggi-X pettorali in una prima esperienza e altre immagini di risonanza magnetica cerebrale in una seconda esperienza. Riuscire ad estrarre informazioni utili dalle immagini non è affatto facile e spesso la possibilità di codificare al meglio l'informazione è inibita da fattori quali un errore sul modello o una scorretta acquisizione dell'immagine. Ciò che permette di estrarre le informazioni dalle immagini sono le feature delle stesse, identificando oggetti nell'immagine che risultano avere caratteristiche comuni. Schemi ricorrenti nei soggetti malati (o anche nei soggetti

sani) permettono di sviluppare criteri oggettivi per determinare la diagnosi dei pazienti ricercando tali schemi.

Come nel caso del dataset del Brain Tumor utilizzato nella tesi, nelle diverse categorie di tumore è sicuramente necessario riconoscere l'adenoma a seconda della sua forma e della variazione di colorazione con cui la massa tumorale si presenta. Dal riconoscimento di determinate caratteristiche è possibile classificare un oggetto mai visto prima dal sistema attribuendogli un classe di appartenenza o label definita a priori. In questo lavoro di tesi il processo di classificazione si è basato su un approccio supervisionato definito nel paragrafo 2.1. Occorre andare ad estrarre all'oggetto in esame, nel nostro caso a immagini di risonanze magnetiche, un certo numero di caratteristiche che ci permettano di identificarlo e la rete neurale utilizzata andrà a farlo direttamente da sola.

Nei problemi di classificazione supervisionata come quello di interesse occorre costruire un classificatore in grado di apprendere dai dati in ingresso e riuscire così a classificare anche un dato mai visto e del quale si vuol conoscere la classe di appartenenza. Lo scopo principale delle nostre reti usate per la classificazione mediante un approccio di apprendimento supervisionato è quello di definire delle buone regole di decisione per l'assegnazione delle etichette ad oggetti sconosciuti, sulla base delle conoscenze che il modello ha acquisito. Per una generica osservazione y , ed un numero di classi k , il classificatore può essere definito dalla funzione:

$$C(y) : R^n \rightarrow 1 \dots k$$

Se y appartiene alla classe j e $C(y) \neq j$, si dice che il classificatore C commette un errore nella classificazione di y . Chiamati I l'insieme dei dati di input e O l'insieme dei dati di output, si definisce una funzione h tale che associa ad ogni dato di ingresso I la sua risposta corretta. Questa funzione h non si conosce ed è quella che va appresa. L'accuratezza di un classificatore C indica la porzione di elementi correttamente classificati su un gruppo di dati N_t con classe nota rapportata ai totali classificati. Dunque la porzione di dati non correttamente classificati sarà: $\delta E = \frac{N_{err}}{N_t}$.

Abbiamo quindi descritto le reti neurali come una semplice catena di strati caratterizzata dalla profondità del modello e dalla larghezza di ogni strato. Variando questi parametri si possono ottenere un numero considerevole di architetture diverse. Molte architetture sono state sviluppate per compiti specifici e nel presente lavoro di tesi

SEZIONE 2.5. CLASSIFICAZIONE

saranno approfondite quelle che sono state utilizzate nella fase sperimentale per la classificazione: le CNN (Convolutional Neural Networks). Nel capitolo successivo viene una piccola digressione su modello matematico di tali reti.

3. Reti neurali convoluzionali

3.1 Funzionamento generale

Le Convolutional Neural Networks [7], o reti di convoluzione, sono reti specializzate nell'elaborazione di dati che hanno la forma di vettori multipli con una topologia nota a forma di griglia. Esse nascono nel 1990 dalla ricerca di Yann LeCun insieme al suo team basandosi sul funzionamento della corteccia visiva del cervello umano. Grazie alle ottime prestazioni che si sono riuscite a ricavare soprattutto in ambito del riconoscimento di immagini, ancora oggi le CNN sono considerate lo "stato dell'arte" per quanto riguarda riconoscimento di pattern ed immagini.

Un'immagine può essere vista come una griglia di due dimensioni di pixel contenente i tre valori di intensità per i tre canali del colore (RGB). L'immagine digitale letta dalla libreria OpenCV può essere vista come una matrice tridimensionale di valori compresi tra 0 e 255. Il nome rete di convoluzione è dato appunto dal fatto che durante il suo funzionamento esegue un'operazione matematica chiamata convoluzione.

Un aspetto chiave di ogni algoritmo di Machine Learning è quello di riuscire ad estrarre i tratti più importanti dai dati in ingresso. Le CNN sono in grado di capire in maniera automatica i tratti più rilevanti e di apprenderne i pattern [15]; per questo motivo di norma gli strati delle CNN vengono considerati come dei blocchi per rilevare i tratti: i primi strati (quelli subito dopo lo strato di ingresso) sono considerati "low-level features extractors", mentre gli ultimi strati (di solito completamente connessi come quelli delle Reti Neurali non convolutive discusse all'inizio) sono considerati "high-level features extractors".

A ciascuna immagine di addestramento vengono applicati dei filtri a diverse risoluzioni e l'output di ciascuna immagine convoluta viene utilizzato come input per il layer successivo. I filtri possono essere inizialmente feature molto semplici, ad esempio la luminosità o i bordi (low level features), e diventare sempre più complessi fino a includere feature che definiscono in modo univoco l'oggetto (high level features). Le CNN elaborano delle *mappe dei tratti* (o *feature maps*) dove ogni elemento corrisponde a dei pixel nell'immagine originale. Per ottenere questo risultato è necessario effettuare appunto l'operazione di convoluzione.

3.2 Architettura generale di una CNN

3.2.1 Convoluzione

Una CNN usa la convoluzione al posto del generale prodotto matriciale in almeno uno dei suoi strati. La convoluzione è una operazione su due funzioni a valori reali; dette queste due funzioni x e w , il loro prodotto di convoluzione sarà

$$s(t) = \int x(a)w(t-a)da = (x * w)(t).$$

Nelle reti di convoluzione spesso la funzione x si riferisce alla funzione di ingresso e la funzione w al kernel, che può essere visto come una funzione di peso relativa ai dati di input. Più in generale, nelle applicazioni l'input è un vettore multidimensionale dei dati e il kernel è un altro array multidimensionale di parametri che vengono adattati dall'algoritmo di apprendimento in maniera appropriata. Per utilità pratiche conviene definire nello specifico l'operazione di convoluzione discreta, un prodotto di convoluzione che invece di essere implementato su un integrale esteso all'infinito è implementato su una sommatoria su un numero finito di indici riferiti agli elementi dei vettori. Per esempio, avendo come input una immagine bidimensionale I , si potrà usare un kernel bidimensionale W tale che:

$$S(i, j) = (I * W)(i, j) = \sum_m \sum_n I(m, n)W(i - m, j - n).$$

Spesso nelle implementazioni software si preferisce utilizzare una funzione che si basa su quella sopra, chiamata cross-correlation, che è la stessa della convoluzione ma con il kernel capovolto.

$$S(i, j) = (I * W)(i, j) = \sum_m \sum_n I(i + m, j + n)W(m, n).$$

Sotto si può vedere una figura che mostra un esempio di feature che un filtro di convoluzione è in grado di estrarre (Figura 3.1).

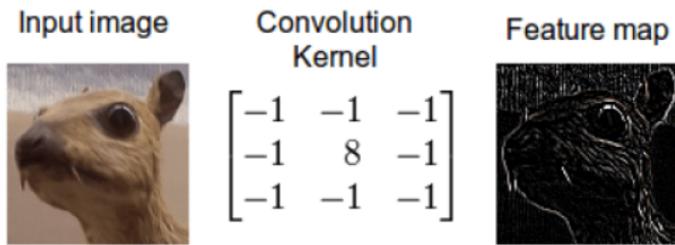


Figura 3.1: Questo kernel è in grado di identificare i bordi della sagoma dell'animale nell'immagine tramite una precisa disposizione dei valori dei pesi nella matrice stessa. [16]

Vi sono due concetti alla base del funzionamento delle CNN :

1. *Sparse Connectivity* (connessioni locali) [17]; l'idea è che, in un'immagine in cui i pixel sono spazialmente più vicini, questi siano più propensi a rappresentare qualcosa di correlato tra di essi. Per esempio, nell'analisi di una immagine di input che abbia centinaia o milioni di pixel siamo interessati a individuare feature significative o dettagli importanti molto piccoli rispetto alla dimensione totale dell'immagine, contenuti per esempio in una decina di pixel come ordine di grandezza. Gli strati tradizionali delle reti neurali usano il prodotto matriciale tra matrici di parametri che descrivono l'interazione tra ogni unità di input con ogni singolo output; questo significa che ogni unità di output interagisce con ogni unità di input. Le reti di convoluzione invece possono avere connessioni locali tra i neuroni che generano interazioni sparse nella rete. Questa connessione locale può essere formalizzata impostando il kernel più piccolo rispetto all'input. Dati m input e n output, un

prodotto matriciale richiederebbe $m \cdot n$ parametri e l'algoritmo una complessità temporale $O(mn)$ di elaborazione. Limitando il numero di connessioni per ogni output ad un numero k più piccolo di m sarebbero richiesti solo $k \cdot n$ parametri e un tempo di elaborazione pari a $O(k \cdot n)$. Il guadagno in efficienza diventa estremamente importante con k più piccoli di m di molti ordini di grandezza.

Dunque l'idea della convoluzione è di preservare le relazioni spaziali tra i pixel. Ridurre il numero di connessioni regolarizza la rete e riduce il rischio di overfitting.

2. *Parameter-sharing* [18] (parametri condivisi): l'idea di base è che gli stessi pesi vengano utilizzati per diversi gruppi di pixel dell'immagine principale.

In una rete tradizionale, ogni elemento della matrice dei pesi è usato esattamente una volta durante il calcolo di un output dello strato e poi non viene più riutilizzato. Qui gli stessi pesi vengono utilizzati per diversi gruppi di pixel dell'immagine principale. Il valore di un peso applicato ad un input è collegato al valore di un peso applicato in un altro punto della rete. In una CNN ogni kernel utilizzato è fatto in modo tale da essere invariante alle traslazioni, quindi è usato in tutte le posizioni dell'input (eccetto che per alcune particolari condizioni al contorno). La rete apprende quindi solo da un determinato set di parametri invece che da tanti set separati per ogni sezione dell'input, il che riduce significativamente il numero di accessi in memoria. Inoltre si sa che così come i neuroni della corteccia cerebrale sono designati a riconoscere feature locali dell'immagine (come angoli o bordi), si utilizzano gli stessi kernel (o filtri) per differenti parti dell'immagine, sempre con gli stessi pesi e valori di attivazione.

I neuroni degli strati di convoluzione sono organizzati nelle cosiddette *feature maps*, nelle quali ogni unità è connessa ad una porzione locale della mappa allo strato seguente attraverso un insieme di pesi. Ogni feature map corrisponde ad un determinato *kernel* che ha il fine di ricercare una certa feature in determinati punti dell'immagine. Data la definizione di convoluzione discreta descritta sopra, dato un filtro di dimensione $k \times k$, con dei pesi associati atti a ricercare una determinata feature, e chiamato θ il valore del bias si fa traslare il filtro lungo l'immagine di input e si fa una somma pesata e shiftata rispetto a θ salvando il risultato ottenuto nelle feature maps stesse, che saranno tante quanti sono il numero di filtri applicato allo strato di input.

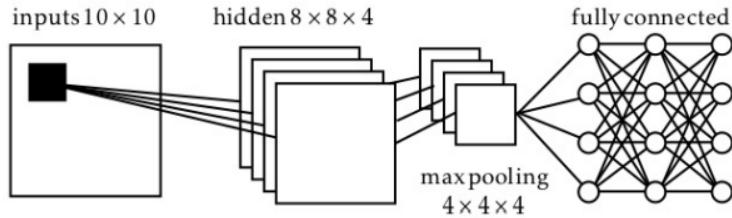


Figura 3.2: Rappresentazione schematica di una CNN. Da sinistra verso destra: uno strato di input, le feature maps ottenute con la convoluzione, gli strati di max-pooling e infine la parte completamente connessa. [7]

Il kernel viene shiftato ogni volta lungo la mappa dello strato precedente in orizzontale e verticale di un valore s , detto *stride*, che è un iperparametro molto importante nella fase di convoluzione.

La formula di convoluzione discreta scritta sopra è utilizzata per un array di due dimensioni. Per immagini colorate di solito si hanno 3 canali per colore, dunque in tale caso l'array in input sarebbe tridimensionale. Si ottengono quindi array di dimensioni superiori a 2, chiamati *tensori*. Tutti i neuroni di uno stesso strato convoluzionale hanno lo stesso valore del bias. Il package software Tensorflow utilizzato nella fase sperimentale è attualmente in grado di eseguire efficientemente tutte le operazioni con i tensori.

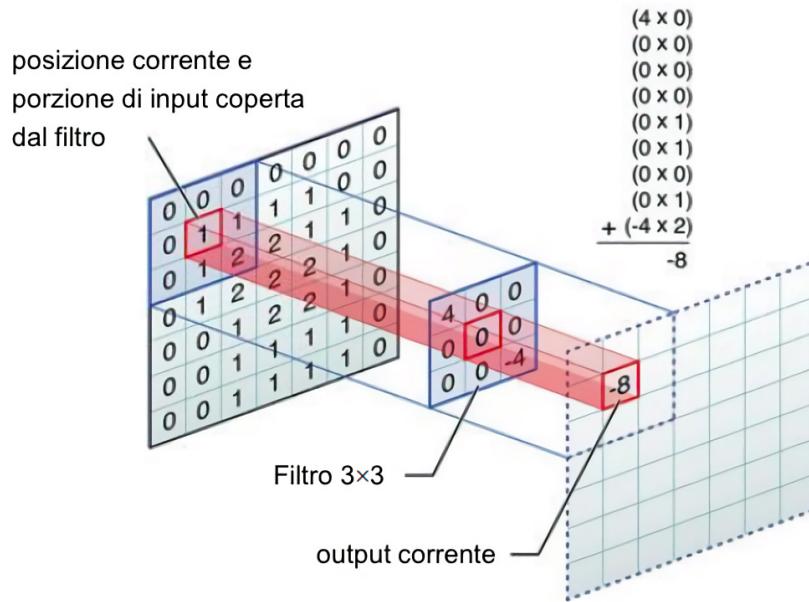


Figura 3.3: Schema di come avviene la convoluzione. [15]

3.2.2 Padding

Se si accoppiano più strati di convoluzione l'uno dopo l'altro, a mano a mano che ci si muove verso destra il numero di neuroni decresce molto velocemente. E' per questo che è necessario il padding, che permette di aggiungere righe o colonne di peso pari a 0, e che permettono l'utilizzo di filtri non troppo piccoli senza dare problemi con le dimensioni in uscita. Esistono 3 tipi di padding ma quello utilizzato di più nell'implementazione di una CNN è il *same padding*, il quale fa in modo di preservare le dimensioni dell'input originale.

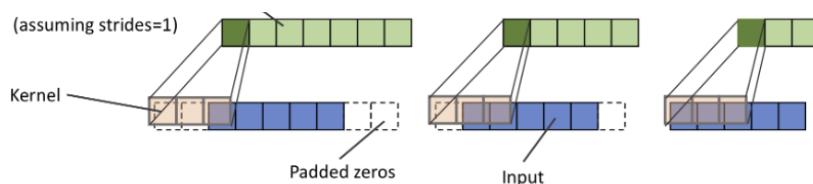


Figura 3.4: Schema dei vari tipi di padding. A sinistra il padding di tipo *full*, al centro quello di tipo *same*, a destra il padding *valid* [12]

Il sistema si calcola automaticamente il valore p di padding da adottare affinché ciò possa essere rispettato. La dimensione della feature map risultato di una convoluzione è data da:

$$o = \frac{(n + 2p - m)}{s} + 1$$

dove:

- n è la dimensione del vettore di ingresso (se è un'immagine si fa sempre in modo di renderla quadrata $n \times n$);
- p , come visto in precedenza è il valore di padding;
- m è la dimensione del filtro;
- s è lo stride.

Da qui è semplice ricavare p per il same padding e/o s per avere o del valore desiderato.

3.2.3 Funzioni di attivazione

Come succede nelle reti neurali artificiali dense, anche nelle convoluzionali è necessario che, una volta fatta la convoluzione e generate le feature maps, i neuroni vengano passati attraverso una funzione di attivazione. Quella più utilizzata negli strati nascosti dei modelli nelle prove sperimentali è la ReLu (Rectified Linear Unit). Con questa le feature maps sono passate attraverso una funzione che ritorna zero se il valore in input è inferiore a 0, mentre ritorna il valore dell'input stesso se questo è maggiore di zero.

Ciò è utile in quanto lo scopo di una CNN è quello di incrementare sempre di più la non linearità del dataset. Quando si guarda ad un'immagine in effetti si nota come essa contenga molte zone non lineari come bordi, angoli, transizioni di colore...

Lo scopo è quello di eliminare i pixel neri così da mantenere solo i grigi e i bianchi rendendo la color transition più bruta ma più efficace al fine di identificare feature.

La ReLu permette dunque di non attivare tutti i neuroni allo stesso momento, riducendo il costo del processo di backpropagation necessario ad una corretta determinazione dei pesi durante il processo di training della rete, che è sicuramente più veloce se si procede in questo modo. La convergenza della rete viene velocizzata di circa 6 volte di più rispetto a utilizzare una funzione di attivazione come la sigmoide o la tangente iperbolica.

La ReLu verrà utilizzata anche perchè risolve in parte il problema della *scomparsa del gradiente* [19], uno dei principali problemi del deep learning. Durante la fase di backpropagation i pesi degli strati in prossimità dell'input restano costanti o si aggiornano molto lentamente al contrario di quanto accade per gli strati vicini all'output. Questo può provocare un rallentamento della rete ed è dovuto alle funzioni di attivazione. Funzioni di attivazione come la sigmoide infatti sono funzioni a codominio limitato e hanno una derivata che presenta una regione di significatività (ovvero la regione del dominio dove la funzione derivata assume i valori più significativi) piuttosto piccola, oltre la quale il valore della derivata stessa è prossimo allo 0. Per la regola della catena nella fase di backpropagation è necessario andare a fare un prodotto di derivate che è pari al numero di strati della rete neurale, come detto nella sezione 2.3. A mano a mano che però si torna indietro dall'output verso l'input però, moltiplicando cifre molto vicine allo zero tra di loro si ottengono valori di aggiornamento dei pesi infinitesimi che comportano l'impossibilità per la rete di apprendere bene, soprattutto quando si vanno ad utilizzare reti molto profonde, e quindi quelle tipiche del DL.

3.2.4 Strati di subsampling: Pooling Layers

[20] Mentre il ruolo dello strato di convoluzione è quello di trovare congiunzioni locali tra feature dello strato precedente, il ruolo dello strato di pooling è quello di fondere semanticamente feature simili in uno solo riducendo la dimensione dei dati. Questo strato di solito è applicato subito dopo quello di convoluzione perchè lavora sull'output processato da questo strato e riprende il concetto di sparse connectivity.

Un neurone di uno strato di pooling prende gli input di diverse feature maps vicine tra loro e li riduce ad un singolo output. Esistono due tipi di subsampling layers (strati di sottocampionamento) di questo tipo:

1. *Max Pooling*: le unità di questo strato calcolano il massimo di una porzione locale (definita a seconda della dimensione stabilita dal codice, ad esempio porzioni 2x2) in una mappa di feature. Allora una funzione di pooling sostituisce l'output di una rete ad una certa locazione con un risultato che riassume quelli ottenuti dagli output vicini, riducendo la dimensione della rappresentazione e focalizzando l'analisi solo sui punti salienti dell'oggetto analizzato.

2. *Mean Pooling*: in questo caso ogni neurone prende il valore medio dei valori della porzione di input.

In sintesi, un livello di pooling esegue un'aggregazione delle informazioni nel volume di input, generando feature maps di dimensione inferiore, conferendo invarianza rispetto a semplici trasformazioni dell'input, mantenendo al tempo stesso le informazioni significative ai fini della discriminazione dei pattern contenuti in essi.

Se si ha una feature map di dimensione $W \times W \times D$, un kernel di pooling di dimensione F e un valore di stride pari a S , allora la dimensione dell'output sarà determinato dalla formula

$$\frac{(W - F)}{S} + 1$$

Il pooling funziona sull'idea di base per cui piccoli cambiamenti non cambieranno il risultato finale, perciò aggiunge robustezza ai dati.

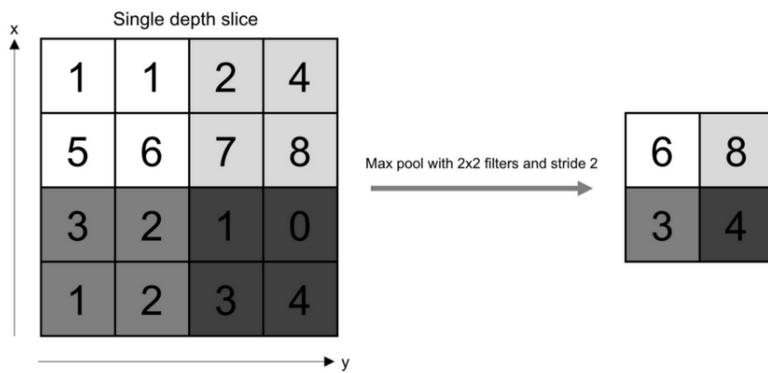


Figura 3.5: Esempio di pooling tramite una pool 2x2. [20]

3.2.5 Fully connected layers

Dopo gli strati di convoluzione/pooling si inseriscono uno o più strati di neuroni completamente connessi (Dense Layers), dopo aver compito un'operazione di Flattening di ogni feature map bidimensionale dello strato precedente. I FC layers connettono tutti i neuroni del livello precedente con quelli del successivo al fine di stabilire le varie classi identificative visualizzate nei precedenti livelli secondo una determinata probabilità. È grazie a questi strati che è possibile effettivamente poi ottenere un vettore in uscita che

SEZIONE 3.2. ARCHITETTURA GENERALE DI UNA CNN

produca una stima probabilistica e che vada ad avere dimensione $c \times 1$, dove c è il numero delle unità di classificazione.

Ricapitolando, l'architettura di una CNN consiste generalmente di questi strati:

- Input
- Convoluzione
- Attivazione non lineare
- Pooling
- Flattening
- Dense layers
- Output

Nella fase sperimentale del lavoro ne sono però stati utilizzati anche altri che saranno discussi in seguito.

4. Ambiente di lavoro

In questa sezione si analizzano i principali strumenti software utilizzati per lavorare con le reti convoluzionali per la classificazione di immagini.

4.1 Python

Per l'implementazione e la sperimentazione delle tecniche e dei modelli di deep-learning il linguaggio scelto è stato Python (v. 3.8.5), per la sua semplicità e versatilità.

La piattaforma utilizzata per lavorare con Python è Anaconda, di grande popolarità nell'ambito della data science in quanto semplifica sensibilmente il processo di setup di un ambiente di sviluppo racchiudendo assieme nella stessa distribuzione l'installer di Python, un package e environment manager dedicato chiamato *Conda* e molte altre librerie integrate. Le principali, usate nel presente lavoro di tesi sono [21]:

- *NumPy*, cioè il package fondamentale per la computazione numerica.
Fornisce oggetti multidimensionali ad alta performance chiamati array, degli strumenti per lavorare con essi in maniera efficiente ed è dotata di funzioni che facilitano le operazioni. È estremamente utilizzato nell'analisi dei dati e nel lavoro di tesi è risultato essere essenziale per lavorare in sinergia con librerie come *TensorFlow*. Permette di creare array N-dimensionali ed è la base di altre librerie come *scikit-learn*;
- *Scikit-learn*, è un package di machine learning che fornisce molti algoritmi e può essere utilizzato anche per problemi di classificazione. Principalmente è stato

impiegato per realizzare i grafici e le strutture utili per mostrare le previsioni del modello (matrici di convoluzione, report di classificazione...)

- *Matplotlib* è una libreria per il plotting e serve a produrre schemi e grafici ed è molto utilizzato in generale per la visualizzazione dei dati a basso utilizzo di memoria. È stato infatti sfruttato per la visualizzazione delle immagini stesse.
- *Pandas* (Python Data Analysis) serve per l'analisi dei dati e fornisce strutture dati veloci e flessibili;
- *OpenCV* (Open-source Computer Vision Library) è una libreria software multipiattaforma nell'ambito della visione artificiale. Con questa libreria è stato possibile leggere e ridimensionare le immagini.

Le librerie che sono state caratterizzanti l'esperienza di lavoro con le reti neurali sono state TensorFlow e Keras.

4.2 Tensorflow e Keras

Keras [22, 23] è una libreria di alto livello scritta in Python. Si tratta di un software per l'apprendimento automatico e le reti neurali, progettata come un'interfaccia a un livello di astrazione superiore di altre librerie simili usate come *back-end*¹. Una delle più utilizzate dal 2017 è TensorFlow, ovvero una libreria di basso livello, un framework che offre molte API per il machine learning e le intelligenze artificiali, sia di alto sia di basso livello. TensorFlow 2.0, rilasciato nell'ottobre 2019, è l'ultima versione utilizzabile ad oggi. Keras è spesso citato nella letteratura scientifica perché utilissimo per sperimentazioni veloci e poco costose, per la sua modularità ed estensibilità. Inoltre gli oggetti che questa libreria permette di utilizzare rispecchiano in maniera molto intuitiva quello che è il loro utilizzo. Esempi di oggetti Keras sono i *modelli*, i *layer* e gli *ottimizzatori*, i quali non sono altro che le strutture che compongono l'architettura di una CNN di cui si è trattato sopra.

Keras minimizza il numero di azioni dell'utente richieste per casi di uso comune, fornisce messaggi di errore molto chiari per un debug rapido e possiede una documentazione molto estesa. Prende vantaggio dall'impiego di TensorFlow. Questa ultima può essere

¹con questo termine si indica parte del software che elabora i dati generati dal front end, ovvero quella parte che gestisce l'interazione con l'utente o con sistemi esterni che producono dati di ingresso

utilizzata per task differenti, ma ha un focus in particolare nel riuscire ad allenare modelli di deep learning e dargli capacità di inferenza. TF può anche essere definita come una libreria matematica simbolica basata sul dataflow e la programmazione differenziale. È molto utilizzata anche nell'ambito della ricerca da parte di Google.

4.3 Jupyter notebook e GoogleCollab

Il codice che è stato sviluppato per creare il sistema di deep learning è stato fatto girare su Jupyter [24] ovvero un'applicazione client-server che permette di editare e fare il running di documenti notebook tramite il browser web. Jupyter può essere eseguito nel desktop locale senza bisogno di un accesso a internet oppure può essere installato in un server remoto e acceduto tramite internet.

Inoltre l'applicazione Jupyter possiede anche una Dashboard, ovvero un pannello di controllo che mostra i file locali e che permette di aprire documenti notebook o fare uno shutdown dei loro kernel. Un kernel di un notebook non è altro che una *computational engine* che esegue il codice contenuto in un documento notebook. Il kernel utilizzato nella fase di scrittura del codice è *ipython*, che esegue appunto codice Python.

A seconda del tipo di operazioni che deve eseguire, il kernel consuma molta CPU e RAM e l'ultima non viene rilasciata fintanto che non si fa lo shutdown del kernel stesso.

Jupyter è molto comodo in quanto permette di decidere a piacimento quando accendere o spegnere il kernel, di salvare i modelli allenati sul disco e riutilizzarli quando si vuole, lasciare il modello fare training anche per ore. Ciononostante l'utilizzo della CPU e della RAM è piuttosto elevato. La CPU con cui sono stati fatti i training su Jupyter è una AMD Ryzen 7 4800H.

Per fare training e impiegare meno tempo e risorse è stato introdotto anni fa l'uso della GPU, il quale velocizza moltissimo il training.

Al fine di evitare di installare altre dipendenze per far uso della GPU su Jupyter, è stato utilizzato in alternativa ad esso il notebook Google Colaboratory, che non sono altro che blocchi note Jupyter ospitati da Collab. Permette di scrivere ed eseguire codice Python nel browser con i seguenti vantaggi:

- nessuna configurazione necessaria;

SEZIONE 4.3. JUPYTER NOTEBOOK E GOOGLECOLLAB

- accesso gratuito alle GPU di Google (ovviamente per un periodo di tempo limitato);
- condivisione semplificata del notebook.

Ovviamente però vi sono dei limiti: le variabili salvate su notebook Collab hanno durata massima di 12 ore e il tempo di runtime del codice scade dopo 90 minuti di inattività.

5. Implementazione delle reti e prove sperimentali

5.1 Obiettivo

L'obiettivo del lavoro svolto è stato quello di utilizzare dei Dataset di immagini biomedicali e sfruttarli per allenare una rete neurale convoluzionale il cui scopo è la classificazione.

Nel dettaglio, in una prima parte dell'esperienza è stato utilizzato un dataset di radiografie pettorali per addestrare un modello che riconoscesse quali tra i soggetti presentassero la polmonite e quali no.

Nella seconda parte si utilizza un dataset di risonanze magnetiche cerebrali per stimare una diagnosi dello stato del paziente tra 4 possibili situazioni: paziente sano, paziente con glioma, meningioma o tumore ipofisario.

5.2 CNN per la rilevazione della polmonite

5.2.1 Il dataset

Il dataset utilizzato, pubblicato sulla piattaforma Kaggle¹ da Paulo Breviglieri, è costituito da 5856 radiografie. È una versione rivisitata di un altro dataset le cui immagini sono state selezionate da uno studio di coorte retrospettivo² di pazienti pediatrici di età che va da 1 a 5 anni di un centro medico di Canton (Hong Kong). Le radiografie sono state realizzate come quadro clinico di routine per i pazienti. Le immagini stesse sono state sottoposte a screening per il controllo della qualità e tutte quelle illeggibili sono state rimosse. Le diagnosi sono poi state confermate da due fisici esperti e successivamente da un terzo specialista prima di essere utilizzate per allenare sistemi di AI.

Il modello di CNN utilizzato dovrà essere in grado di riconoscere i pattern di immagine tipici della malattia. L'identificazione è infatti talvolta complicata anche per i medici più esperti.

Sotto: esempio illustrativo di raggi-X in pazienti con polmonite.

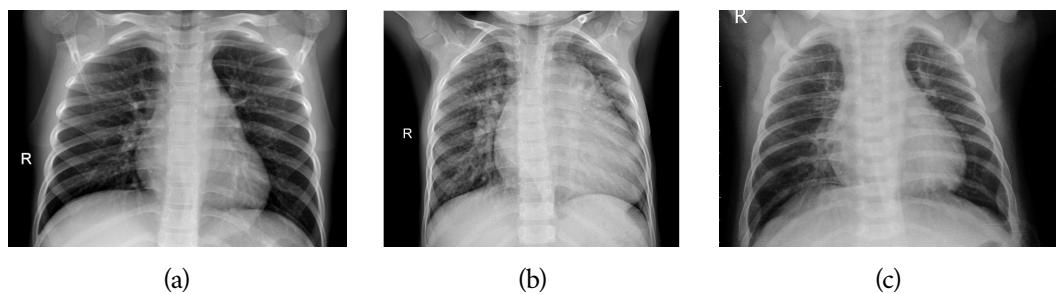


Figura 5.1: Si può notare come un soggetto sano (a) mostri polmoni senza aree di anomale opacità. L'immagine (b) è invece un caso di polmonite batterica che presenta il tipico consolidamento. (c) è un caso di polmonite virale che presenta un'un'opacità interstiziale più diffusa in entrambi i polmoni.

¹Kaggle è una piattaforma online per competizioni di modelli predittivi e analitici fondata nel 2010. Ad oggi conta più di 500.000 utenti

²Uno studio di coorte prende in considerazione un gruppo di individui che presentano caratteristiche comuni (sesso, età, etnia...) e che hanno come unica differenza tra loro l'esposizione o meno al fattore di rischio. Questo tipo di studio identifica persone con una malattia (o altre variabili di interesse) e le paragona ad un gruppo di controllo appropriato che non presenta la patologia.

Questo gruppo viene osservato per un periodo di tempo prestabilito, al termine del quale si analizzerà la presenza o meno dell'esito atteso.

Il dataset [25] è organizzato in 3 cartelle: una di Training (per l'addestramento stesso della rete), una di Testing (per testare le capacità di generalizzazione della rete) e una di Validation (per testare la bontà della rete e rilevare overfitting e/o underfitting in fase di addestramento). Ognuna di esse è organizzata così:

- 4,192 immagini per il training (1,082 casi normali, 3,110 di polmonite);
- 1,040 immagini di validation (267 casi normali, 773 di polmonite);
- 624 immagini per il testing (234 casi normali, 390 di polmonite).

Il numero di immagini è sufficientemente grande per poter essere utilizzato per gli obiettivi preposti ricorrendo alle tecniche di deep learning per la computer vision di ambito biomedico.

Le proporzioni del dataset rispettano ampiamente gli standard che di solito si utilizzano per addestrare un modello di classificazione: un set di testing e uno di validation che rappresentano ognuna il 10% del totale, il resto è lasciato al training. Si va adesso a sintetizzare quali sono stati i principali passaggi per l'implementazione della CNN e il suo addestramento.

5.2.2 Setup iniziale

Il primo step per realizzare il classificatore è l'importazione delle librerie necessarie per la manipolazione di tensori, per le operazioni matematiche e per la relizzazione di grafici (Pandas, NumPy, Matplotlib), così come librerie di Keras per l'image preprocessing e oggetti di Deep Learning, tra cui:

- `Sequential`, classe di Keras che permette di raggruppare layers, che non sono altro che i blocchi di base per la costruzione delle reti neurali, in uno stack lineare e realizzare così un oggetto di classe `Model`, che è implementato in modo tale da riuscire a elaborare deduzioni una volta superata la fase di training;
- `Conv2D`, classe che realizza la convoluzione spaziale sulle immagini 2D. Questo strato sfrutta un kernel di convoluzione il quale è coinvolto con lo strato in input per produrre un tensore in output. Se questo strato è usato per primo nel modello è necessario inserire come parametro la tupla rappresentante le dimensioni delle

immagini in input. In questo strato è possibile scegliere il numero e le dimensioni del filtro, il valore dello stride (s=1 di default), il tipo di padding e altri parametri del caso;

- `MaxPool2D`, realizza l'operazione di pooling, dunque è possibile anche qui scegliere la dimensione della pool, il valore dello stride, il tipo di padding...;
- `Flatten`, classe che permette di trasformare la feature map allo strato precedente in un vettore che viene dato in input alla ANN in seguito;
- `Dense`, classe che implementa un regolare strato di una NN. In sostanza è questo lo strato che implementa la somma pesata degli input con i pesi e il bias di cui si parlava nel capitolo 2.

5.2.3 Definizione degli iperparametri

Successivamente sono stati fissati dei valori per gli iperparametri, ovvero quelle variabili poste all'inizio del codice, prima che il processo di apprendimento cominci e che possono essere modificate a piacimento fintanto che si trovano quelle che portano ai risultati migliori. I valori degli iperparametri sono davvero importanti per l'accuratezza del modello e delle predizioni, infatti sono stati fatti variare alcune volte prima di arrivare alla migliore soluzione. I principali sono:

- Le dimensioni di input dell'immagine, perchè è necessario che durante la fase di training vi sia un'uniformità delle dimensioni degli input;
- il numero di epoch, cioè il numero di volte in cui l'algoritmo di apprendimento lavora sull'intero dataset prima di procedere con l'aggiornamento dei pesi. Un'epoca descrive il numero di volte che l'algoritmo vede il intero set di dati. Sulla base di queste varia anche la durata dell'apprendimento. Infatti gli step di apprendimento per epoca sono pari al rapporto tra il numero di istanze del set di training e la batch size;
- la batch size: è un iperparametro dell'algoritmo del gradiente discendente che indica il numero di campioni di training sui quali lavorare prima che venga fatto un aggiornamento dei pesi. Sia in questa esperienza sia in quella successiva si utilizza

un algoritmo di apprendimento in modalità *mini-batch* in quanto il dataset è abbastanza numeroso, ovvero si utilizza un valore di batch più grande rispetto a 1, così che non si vada ad aggiornare i valori dei pesi ogni volta che il sistema riceve un campione in input, quindi più del necessario, ottimizzando la durata dell'epoca e quindi del training. Di norma la batch size deve essere un valore che sia divisore del numero totale di immagini del dataset e nelle folders.

- il numero di feature maps: più che ne sono più caratteristiche differenti si possono andare a scovare ma allo stesso tempo aumenta anche il numero di parametri e se si vuole evitare il rischio di overfitting occorre aumentare la complessità del modello (ad esempio aggiungendo più strati senza perdita di troppe informazioni).
- il numero di canali d'immagine utilizzati nel processo di learning: per le immagini RGB colorate tale parametro è pari a 3, mentre nel caso di immagini in scala di grigi vale 1. In tale caso le immagini sono digitali RGB ma vedremo che utilizzare solo un canale è la soluzione che porta ad una maggiore accuratezza in questa prima esperienza.

Facendo riferimento agli iperparametri utilizzati per la costruzione del classificatore sono stati scelti tali iperparametri come i migliori per questa esperienza:

```
img_height = img_width = 600
epochs = 20
batch_size = 8
hyper_featuremaps = 32
hyper_channels = 1
hyper_mode = 'grayscale'
```

E' stata utilizzata una dimensione per le immagini di 600 x 600 pixel associata a una batch size di 8 affinchè lo spazio occupato in RAM non sia eccessivamente alto.

Un altro iperparametro importante è il learning rate, ma il suo valore viene discusso in seguito.

5.2.4 Definizione e compilazione del modello

Per questa esperienza è stato utilizzato un modello di rete neurale convoluzionale semplice, che consiste di questi strati:

- 5 strati di Convoluzione/pooling: i primi 3 producono ognuno 32 feature maps con la convoluzione, generanti ognuna una feature map in seguito ad un'operazione di max-pooling. Per gli ultimi 2 sono stati invece utilizzati 64 filtri, così da produrre 64 feature maps anch'esse seguite da un'operazione di pooling. La convoluzione è stata performata con dei kernel di dimensione 3x3 e con una funzione di attivazione non lineare, cioè la ReLu. Nel capitolo precedente è stato indicato in parte il perchè questo tipo di funzione è quella maggiormente utilizzata in ambito di classificazione d'immagine. L'operazione di Max-pooling viene fatta con delle pool di dimensione 2x2. Lo stride è stato lasciato quello di default a 1 e il padding scelto è il 'same', affinchè la dimensione dell'input sia uguale alla dimensione dell'output. Viene utilizzato questo tipo di padding affinchè si abbia la certezza che il filtro venga applicato a tutti gli elementi dell'input.
- Strato di flattening: necessario per introdurre la successiva ANN, connessa all'ultimo strato di pooling tramite un unico tensore unidimensionale.
- 3 strati completamente connessi rispettivamente di 128, 64 e 1 neuroni. L'ultimo strato consta di un solo neurone in quanto la classificazione è binaria.

Tipicamente si parte sempre dall'utilizzare un numero di filtri più piccolo, come in questo caso 32, per poi andare ad aumentare a multipli di questa dimensione. Il modello è infine compilato usando il metodo `compile()` sfruttando la funzione di ottimizzazione Adam, algoritmo che si basa sull'idea del gradiente discendente ma che però elabora una stima adattativa dei momenti del primo e del secondo ordine, permettendo una maggiore efficienza rispetto agli altri algoritmi a livello di costo computazionale per il training, a discapito di una conseguente minor capacità di generalizzazione. Inoltre Adam adatta il learning rate a seconda dei vari strati anche sulla base di quello che è il problema della scomparsa del gradiente di cui si è parlato sopra.

Con Adam il learning rate iniziale di default per il training vale 0.001 e tale valore viene

poi ridotto tramite una specifica funzione quando ce ne è stato bisogno.

In fase di compilazione si sceglie anche la metrica, che permette di calcolare quanto spesso le label effettive coincidano con le predizioni fatte ed è dunque un parametro necessario per monitorare l'accuratezza e l'errore del modello. Una metrica di questo tipo viene settata col nome di 'accuracy'. Per il calcolo dell'errore si usa una funzione di costo che in questo caso è chiamata *binary crossentropy*, (funzione di entropia incrociata binaria) poichè si va a fare una classificazione binaria.

```
cnn.add(Conv2D(hyper_featuremaps, (3, 3), activation="relu",
               input_shape=(img_width,
                           img_height, hyper_channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(hyper_featuremaps, (3, 3), activation="relu",
               input_shape=(img_width,
                           img_height, hyper_channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(hyper_featuremaps, (3, 3), activation="relu",
               input_shape=(img_width,
                           img_height, hyper_channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(hyper_featuremaps * 2, (3, 3), activation="relu",
               input_shape=(img_width,
                           img_height, hyper_channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(hyper_featuremaps * 2, (3, 3), activation="relu",
               input_shape=(img_width,
                           img_height, hyper_channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Flatten())
cnn.add(Dense(activation = 'relu', units = 128))
cnn.add(Dense(activation = 'relu', units = 64))
cnn.add(Dense(activation = 'sigmoid', units = 1))
cnn.compile(optimizer = 'adam', loss = 'binary_crossentropy',
            metrics = ['accuracy'])
cnn.summary()
```

Code Listing 5.1: Modello in Python utilizzato

SEZIONE 5.2. CNN PER LA RILEVAZIONE DELLA POLMONITE

Layer (<code>type</code>)	Output Shape	Param #
=		
conv2d (Conv2D)	(<code>None</code> , 598, 598, 32)	320

max_pooling2d (MaxPooling2D)	(<code>None</code> , 299, 299, 32)	0

conv2d_1 (Conv2D)	(<code>None</code> , 297, 297, 32)	9248

max_pooling2d_1 (MaxPooling2D)	(<code>None</code> , 148, 148, 32)	0

conv2d_2 (Conv2D)	(<code>None</code> , 146, 146, 32)	9248

max_pooling2d_2 (MaxPooling2D)	(<code>None</code> , 73, 73, 32)	0

conv2d_3 (Conv2D)	(<code>None</code> , 71, 71, 64)	18496

max_pooling2d_3 (MaxPooling2D)	(<code>None</code> , 35, 35, 64)	0

conv2d_4 (Conv2D)	(<code>None</code> , 33, 33, 64)	36928

max_pooling2d_4 (MaxPooling2D)	(<code>None</code> , 16, 16, 64)	0

flatten (Flatten)	(<code>None</code> , 16384)	0

dense (Dense)	(<code>None</code> , 128)	2097280

```
dense_1 (Dense)           (None, 64)      8256
-----
dense_2 (Dense)           (None, 1)       65
=====
Total params: 2,179,841
Trainable params: 2,179,841
Non-trainable params: 0
```

Code Listing 5.2: Riepilogo del modello utilizzato se in input vi è un immagine con `image_size` pari a 600. La colonna `layer` indica il tipo di strato di cui si tratta. La colonna `output` mostra la tupla con le dimensioni dello strato in uscita (vedasi sezione 3.2.2 e 3.2.3 per calcolarlo), con una dimensione in più `None` che è aggiunta per ospitare la batch size. La terza colonna `Param #` indica il numero di pesi all'interno della rete, i quali possono essere distinti in addestrabili, cioè quelli che vengono aggiornati durante la fase di backpropagation, e quelli per cui questo non vale per motivi di regolarizzazione della rete. Il numero totale di parametri si trova $(\text{kernel_height} * \text{kernel_width} * \text{input_filters} + \text{bias}) * \text{output_filters}$. Ad esempio nel primo strato si avrà $3 * 3 * 32 * 1 + 32 = 320$, in quanto il valore del bias è 1 di default

5.2.5 Creazione dei set di training e validation traminte l'uso dell'image flowing

Tramite l'utilizzo della classe di TensorFlow `ImageDataGenerator()` è possibile andare a creare dei generatori contenenti dati delle immagini di training, testing e validation in forma di tensori andando anche ad apportare modifiche ad esse.

Andando a richiamare su tali generatori il metodo `flowFromDirectory()`, è possibile ritornare un oggetto di tipo `DataFrameGenerator` costituito da una tupla (X, y) dove X è un NumPy array contenente un numero di immagini della folder indicata nel path (che vengono elaborate e rappresentate tramite i valori dei loro pixel) pari alla `batch size` e della dimensione pari a `image_width x image_height` indicata e y è anch'esso un NumPy array che però contiene le label corrispondenti alle immagini

prodotte. Proprio l'oggetto (X, y) è quello che viene dato in impasto alla rete per fare training.

Durante la fase di addestramento viene elaborato il vettore X e per verificare se un risultato è o meno corretto, si compara il dato in uscita dalla rete con quello di y , dove (X, y) è la tupla ottenuta andando a richiamare `flowFromDirectory()` sul path che porta alla directory di training: se sono uguali allora il risultato è corretto, altrimenti il risultato necessita di correzioni. L'utilizzo di questa funzione è stato efficace perché il dataset è strutturato in partenza in training, validation e testing. Inoltre tale funzione genera i set provvedendo a fare uno shuffle di tutte le immagini sia di training sia di validation, mentre per il set di testing è stato inserito `Shuffle = False` così da poter testare la capacità di generalizzazione della rete e confrontare le predizioni con i risultati effettivi, con la possibilità di scegliere a piacimento le immagini di testing su cui provare il funzionamento della rete.

`ImageDataGenerator` è una classe che permette anche di utilizzare tecniche di *augmentation* [26] per ampliare il dataset. Tali tecniche prevedono di andare a costruire versioni modificate delle immagini stesse. Infatti il dataset, pur essendo numeroso, non lo è al punto tale da rendere sufficientemente buona l'esperienza di image processing. Questa mancanza può essere pertanto colmata andando ad ampliare il dataset con immagini che possano arricchire l'esperienza di training del modello. Tali immagini possono essere modificate in vari modi, ma non tutti questi sono utili nel caso di interesse: infatti alcune tecniche possono generare rumore aggiuntivo e "confondere" il sistema nella ricerca dei pattern per la rilevazione della polmonite. Infatti è differente classificare immagini biomedicali come i raggi X per capire se vi è o meno una polmonite rispetto ad una classificazione di immagini come quelli di cani o di gatti. Infatti, mentre un gatto può essere visto da varie angolazioni e ciò può essere utile al fine di distinguerlo da un cane, operare una rotazione troppo elevata per andare a riconoscere l'opacità dei polmoni in un'immagine a raggi X, non va ad arricchire il dataset, ma può portare addirittura a un peggioramento dell'accuratezza del modello. Nell'esperienza è stato fatto un training del modello dapprima senza l'utilizzo di tecniche di image augmentation, per poi confrontarlo con un altro training in cui è stato ampliato il dataset.

Occorre pertanto scegliere accuratamente quali parametri inserire. Le tecniche che sono risultate essere utili in questo caso sono:

- **Rescaling:** dato che è stata definita la modalità di colorazione 'grayscale' ogni pixel di ogni immagine avrà un valore che sta nel range [0,255] che con questa tecnica diviene compreso tra 0 e 1. Un primo beneficio è che così tutte le immagini vengono trattate alla stessa maniera. Infatti è possibile che alcune immagini abbiano un range alto di valori dei pixel, altre più basso, ma entrambe condividono poi lo stesso modello e learning rate per il training. In generale è bene fare in modo che il range di valori sia compreso tra 0 e 1 così che ogni immagine contribuisca nella maniera più uniforme possibile per il calcolo della funzione di perdita totale e quindi per l'aggiornamento dei pesi.
- **Rotazione:** è stato appurato che un piccolo range di rotazione per le immagini possa essere utile a migliorare le capacità di generalizzazione del sistema, proprio perché molto spesso nella pratica clinica le radiografie possono essere lievemente ruotate a causa dei movimenti del paziente. Il range utilizzato in questo caso è tra i -5° e i 5° .
- **Zoom:** come nel caso della rotazione, è prassi che vi siano immagini in cui il torace del paziente sia posizionato più o meno vicino al dispositivo che elabora l'immagine. Quindi un piccolo range di variazione dello zoom delle immagini (in questo caso è stato scelto 0.2) può essere utile.

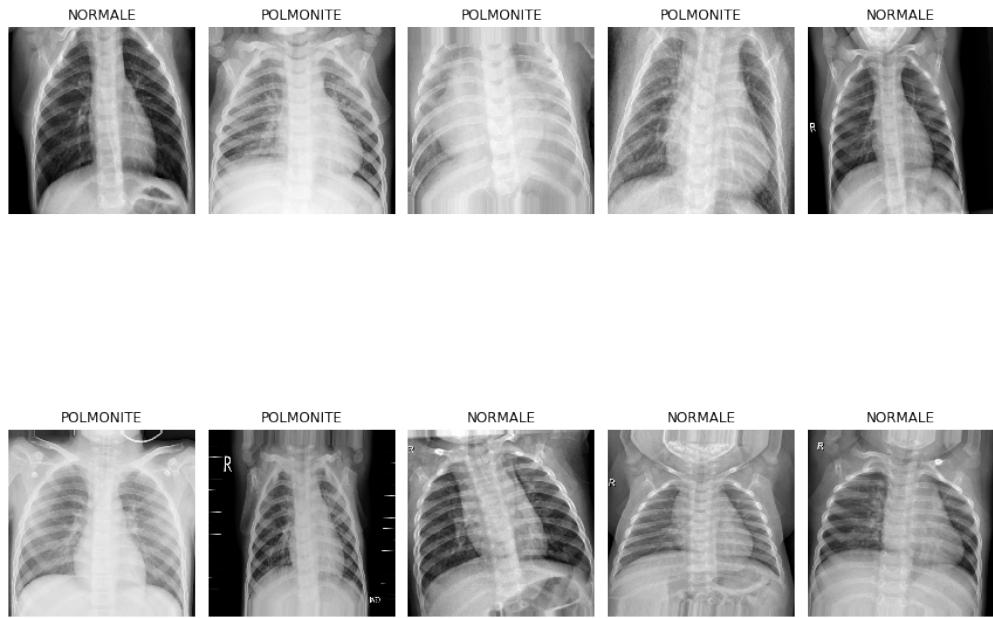


Figura 5.2: Campioni di immagini del dataset di training sui quali sono state aggiunte le tecniche sopra citate. Immagine tratta dal codice (vedasi capitolo 7).

5.2.6 Fase di fitting

Dopo aver compilato e configurato il modello si passa alla fase di *fitting*, cioè la fase di addestramento vera e propria. Il modello viene dunque allenato tramite le immagini del set di training, con un aggiornamento dei pesi che viene fatto per ogni gruppo di campioni pari alla *batchsize*. Il set di training viene rivisto per un totale di 20 epoch, anche se dopo 10 la capacità di training risulta essere stagnante. Una volta captato questo è stata usata la funzione di callback *EarlyStopping()* che ha fermato il training non appena risultava esserci overfitting. Il metodo *fit()* ci permette anche di scegliere l'insieme di validation su cui la rete può generalizzare. Ciò fa sì che non solo si possa monitorare l'accuratezza nel training, ma anche la capacità di generalizzazione della rete, cercando di capire se questa sta andando in overfitting, underfitting o se il trade-off tra le due è accettabile (Figura 5.3), così da agire di conseguenza per poterla migliorare.

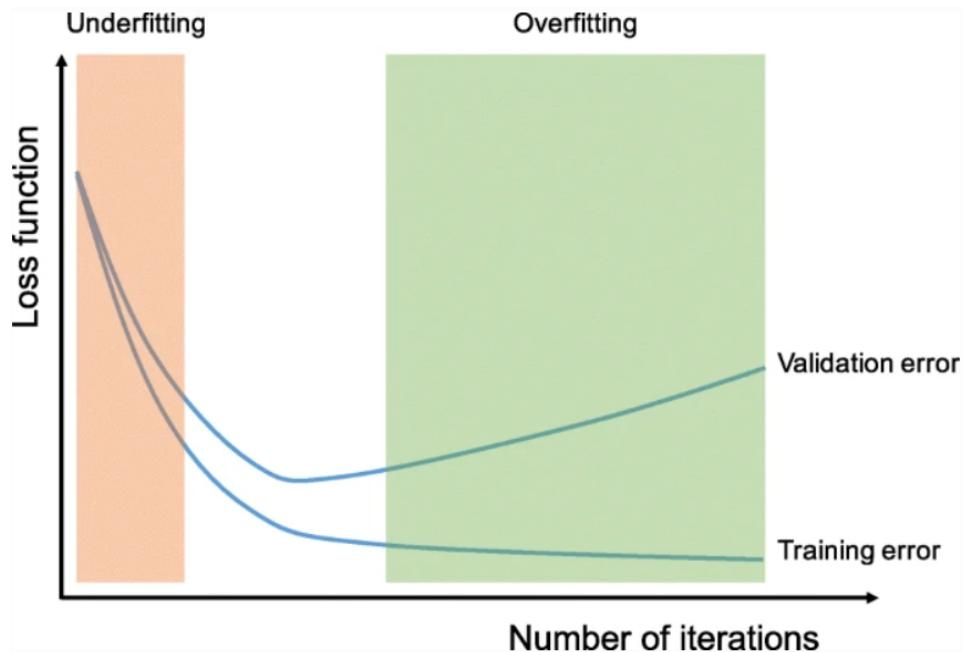


Figura 5.3: Si vede come si va in overfitting quando l'errore di validation è molto maggiore di quello di training all'aumentare delle epoche, mentre in underfitting si va quando non si hanno sufficienti parametri per allenare il modello.[12]

Si può anche definire una lista di chiamate (callbacks) per customizzare il training, come definire un checkpoint dove salvare il modello in formato .h5 così da poterlo utilizzare successivamente per elaborare nuove predizioni, senza necessità di allenarlo nuovamente. Inoltre è stata utilizzata una funzione che riduce il valore del learning rate del 30% automaticamente ogni due epoche in quanto molto spesso il modello beneficia di ciò se l'apprendimento stagna. In questo caso il modello è stato allenato sull'insieme di training e validation definito dal dataset stesso. Il set di testing è l'insieme di immagini sulle quali si va a fare la valutazione finale.

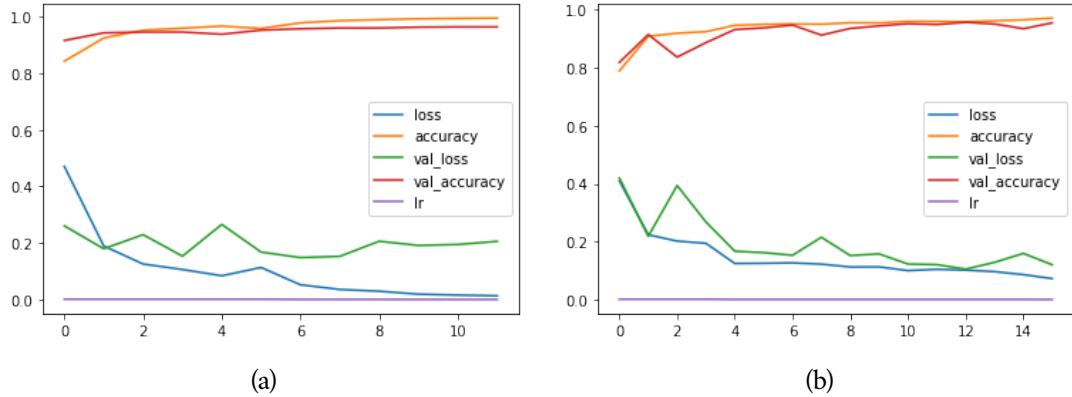


Figura 5.4: In entrambe le figure è possibile osservare le curve che rappresentano accuracy e loss rispettivamente per training e sul set di validation nei due casi. (a) si riferisce al set di training originale, (b) al set di training modificato come nella figura 5.2. Si vede che nel primo caso la funzione di perdita è più alta del secondo caso. Le figure sono state create con la libreria Pandas (codice capitolo 7). Il numero di epoche tra le due immagini è diverso perché nel primo caso la funzione EarlyStopping ha fermato il training alla undicesima epoca, nel secondo alla quindicesima.

Alla fine si ottengono questi risultati andando a richiamare `cnn.evaluate(test)`:

- Per il caso senza modifiche al set di training risulta che l'accuracy sul training è pari al $(99.61 \pm 0.01)\%$ e quella sull'insieme di validation è del $(94.44 \pm 0.01)\%$. Si è impiegato con l'utilizzo della CPU circa 22 minuti a epoca per un totale di 20 epoche (un totale di circa 7 ore) per allenare il modello. Con una GPU si impiegano circa 20 secondi a epoca.
- Per il caso con le modifiche al set di training risulta che l'accuracy sul training è pari al $(96.99 \pm 0.01)\%$ e quella sull'insieme di validation è del $(94.61 \pm 0.01)\%$. Si è impiegato con l'utilizzo della CPU circa 30 minuti per epoca per 20 epoche (un totale di circa 10 ore) per allenare il modello. Con una GPU si impiegano circa 25 secondi a epoca.

5.2.7 Predizioni e grafici

Infine occorre osservare come il sistema riesce a generalizzare sull'insieme di test. Dato che la funzione di attivazione dello strato finale del modello è la sigmoide, il valore dell'output starà in un range compreso tra 0 e 1 corrispondente alla probabilità che l'immagine su cui è

stata fatta la predizione presenti polmonite. Se si vogliono quantificare i falsi positivi e i falsi negativi occorre vedere se l'output della rete è superiore o inferiore a 0.5 (se è superiore o uguale risulta esserci polmonite) e di conseguenza predire se si tratta di polmonite o meno. È possibile visualizzare la matrice di confusione³. Le tabelle 5.1 e 5.2 rappresentano dei report di classificazione in cui è possibile osservare 3 diverse voci:

- Precision = $\frac{TP}{(TP+FP)}$: rappresenta l'accuratezza di una predizione con polmonite, ma non tiene conto dei falsi negativi.
- Recall = $\frac{TP}{(TP+FN)}$: rappresenta il numero di istanze con polmonite correttamente identificate, dunque tiene conto anche dei FN.
- F1 = $\frac{(2*Precision*Recall)}{(Precision+Recall)}$: rappresenta un compromesso tra recall e precision.
- support indica il numero di immagini utilizzate come supporto al calcolo delle previsioni.
- Il valore di accuracy a cui si fa sempre riferimento è $\frac{(TP+TN)}{(TP+TN+FP+FN)}$.

Un modello deve cercare di avere il più possibile un valore di recall superiore allo 0.9 per funzionare bene.

	precision	recall	f1-score	support
NORMALE	0.93	0.29	0.44	234
POLMONITE	0.70	0.99	0.82	390

Tabella 5.1: Classification report per il modello con il set mantenuto come l'originale.

	precision	recall	f1-score	support
NORMALE	0.97	0.86	0.91	234
POLMONITE	0.92	0.98	0.95	390

Tabella 5.2: Classification report per il modello con il set ampliato.

³È un modo per visualizzare in maniera diretta il numero di *True-positives* TP (in questo caso si intendono i soggetti che sono stati previsti avere polmonite e la predizione è corretta), *true-negatives* TN (il contrario), falsi-negativi (FN) e falsi-positivi(FP)

In sintesi l'accuratezza del modello sull'insieme di test vale:

- $(75.80 \pm 0.01)\%$ nella prima esperienza;
- $(93.75 \pm 0.01)\%$ nella seconda.

Dunque sembra che, nonostante nella prima esperienza si abbia ottenuto un'accuratezza migliore sul training, poi le capacità di generalizzazione sono migliori quelle ottenute con la seconda. Ciò significa che nel primo training si è avuto un problema di overfitting.

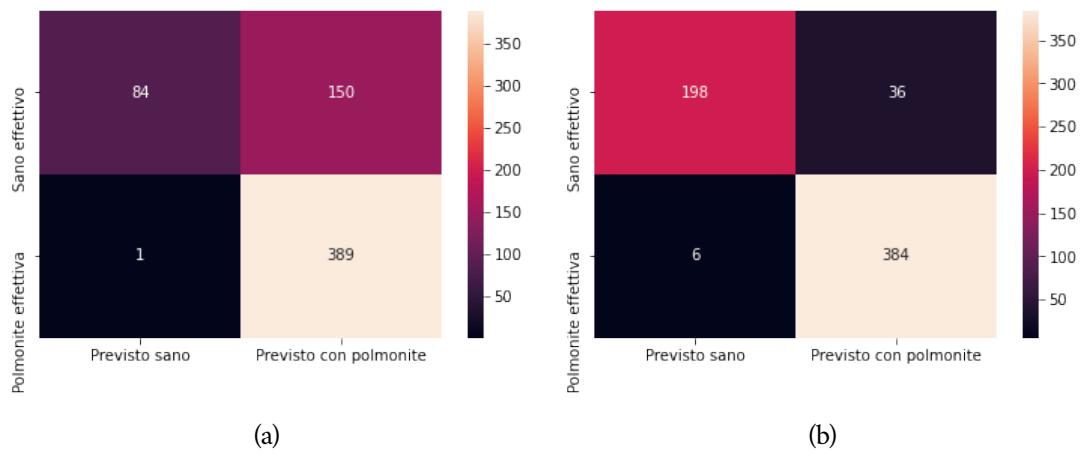


Figura 5.5: Matrici di confusione ottenute. (a) è la matrice di confusione del modello con il set di training lasciato come l'originale. Si può vedere che vi sono molti errori nel prevedere il soggetto sano.

(b) è la matrice di confusione con il set di training modificato come nella Figura 5.2. In questo caso vi sono pochissimi errori di predizione.

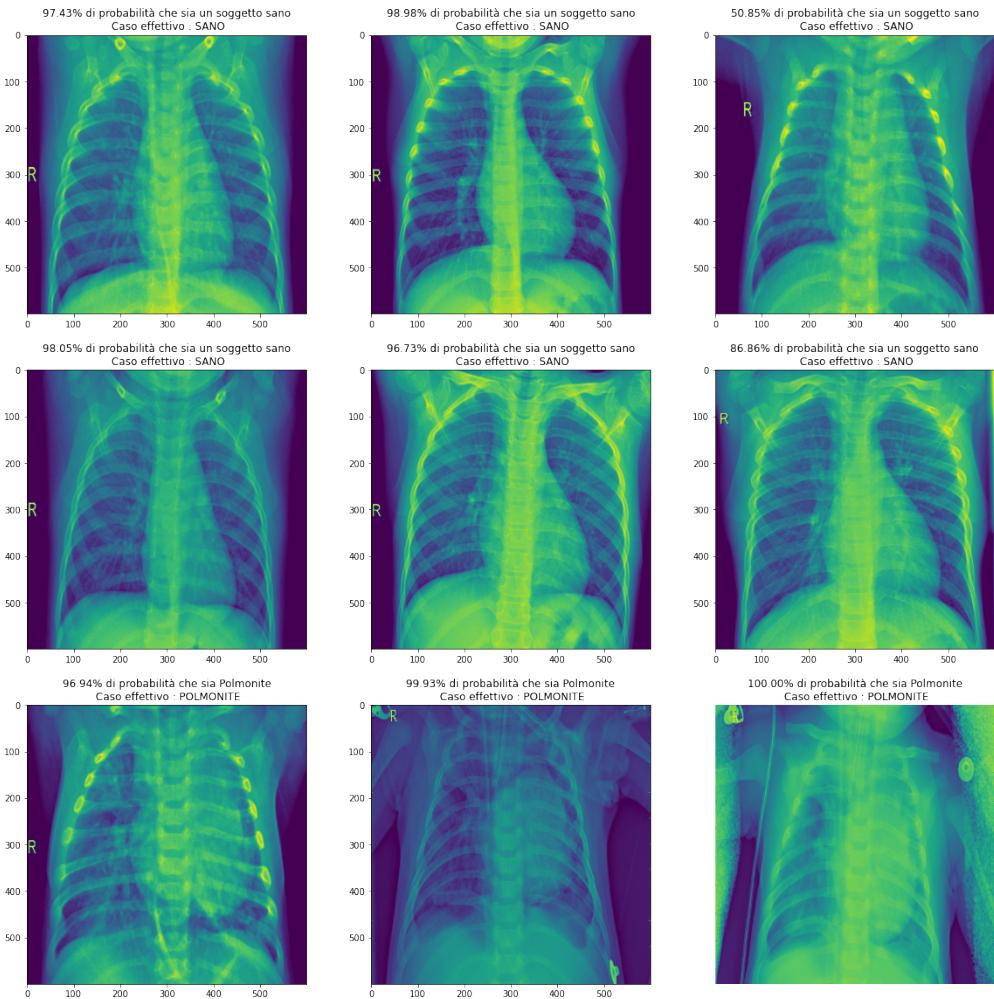


Figura 5.6: Dimostrazione di alcune previsioni sul set della polmonite del sistema allenato con testin accuracy media del 93.75%. (Codice nel capitolo 7).

5.2.8 Considerazioni finali

La rete addestrata con il dataset ampliato risulta produrre buoni risultati. In prima istanza sicuramente grazie all'accuratezza del dataset, ma anche grazie a degli iperparametri ben scelti. In letteratura, la fase in cui si scelgono questi ultimi una volta definito il modello è chiamata *fine-tuning*. Sono state in effetti fatte delle prove con image size di dimensioni 400x400, 500x500, 600x600 e batch size = 8, 16, 32 , ma la migliore accuratezza di generalizzazione si ha con gli iperparametri della sezione 5.2.3. Risultati ragionevoli sono stati trovati dopo 10 epoch, anche se è stato fatto un addestramento anche con 20 e 50

SEZIONE 5.2. CNN PER LA RILEVAZIONE DELLA POLMONITE

epoch , ma ciò risultava inutile in quanto dalla decima epoca in poi l'accuratezza del modello rimane costante e non migliora. Per evitare ciò è stato utilizzato l'Early Stopping che ha appunto fermato il modello intorno alla decima epoca. Inoltre è stato aumentato anche il numero dei canali d'immagine da 1 a 3 per osservare se vi fosse un miglioramento ma ciò non ha fatto altro che peggiorare le prestazioni. Probabilmente questo è dovuto al fatto che per rilevare una polmonite la scala di grigi evidenzia maggiormente l'opacità del polmone. Sicuramente se il classificatore dovesse essere utilizzato in uno studio radiologico, sarebbe un buon mezzo di supporto in quanto il numero di falsi negativi è minore rispetto a quello dei falsi positivi ed è molto piccolo (clinicamente è meglio che sia diagnosticato un falso positivo piuttosto che un falso negativo!). Sotto si mostrano i risultati ottenuti con iperparametri differenti (eccetto il numero di epoch in tutti i casi pari a 20), con le modifiche apportate al set di training della sezione 5.2.5 e che hanno portato alla scelta di quelli alla sezione 5.2.3.

	training accuracy	validation accuracy	testing accuracy
batch = 16 image size = 500x500 channels = 1	95.99%	93.65%	91.98%
batch = 32 image size = 400x400 channels = 1	95.94%	95.00%	91.5%
batch = 8 image size = 600 x 600 channels = 1	96.49%	94.61%	93.75%
batch = 8 image size = 600 x 600 channels = 3	96.32%	93.94%	91.62%

Tabella 5.3: La tabella mostra come i parametri della terza riga abbiano portato ai migliori risultati.

5.3 CNN per la classificazione di risonanze magnetiche cerebrali

Il tumore all'encefalo è considerato una delle patologie più aggressive tra adulti e bambini e rappresenta circa l'85-90% di tutti i tumori che affettano il sistema nervoso centrale. Ogni anno circa 11.700 persone sono diagnosticate con questo tipo di tumore. La percentuale di sopravvivenza per 5 anni almeno dalla diagnosi è approssimativamente del 34% per gli uomini e del 36% per le donne.

La miglior tecnica per rilevare il tumore all'encefalo è la risonanza magnetica (Magnetic Resonance Imaging).

Le RM forniscono immagini cerebrali in cui è possibile evidenziare gli effetti macroscopici delle malattie neurologiche come ad esempio i cambiamenti di forma, intensità e dimensioni della massa tumorale. Un trattamento appropriato e una diagnostica accurata sono fondamentali per un aumento delle aspettative di vita dei pazienti. Un esame manuale da parte del radiologo in questo caso è ancora più complicato rispetto alla rilevazione di una polmonite e molto spesso occorre integrare tali risonanze ad altri esami aggiuntivi. L'applicazione delle tecniche di deep learning hanno mostrato un'accuratezza sempre maggiore anche in questo campo. Vi possono essere molte anomalie sulla grandezza e la posizione del tumore e questo rende difficile capire completamente la natura di questi. Un modello di CNN che riesca a classificare la tipologia di tumore dal semplice riconoscimento di immagine dal canto suo può essere uno strumento molto utile. È stato proprio questo l'obiettivo di realizzazione: implementare una CNN che riconoscesse tra 4 possibili stati del paziente a partire dalla semplice visione della sua risonanza magnetica.

5.3.1 Il dataset

Il dataset [27] è stato anche in questo caso trovato nella piattaforma Kaggle e le 3264 immagini riportano risonanze magnetiche encefaliche dove l'encefalo è visto dall'alto e in piano sagittale e sono raggruppate in 2 cartelle, una di Training e una di Testing. Ognuna di esse è organizzata così:

SEZIONE 5.3. CNN PER LA CLASSIFICAZIONE DI RISONANZE MAGNETICHE CEREBRALI

- nella training folder ci sono 395 immagini dell'encefalo sano, 827 del tumore ipofisario, 822 del meningioma e 826 del glioma.
- nella testing folder ci sono 105 immagini di encefalo sano, 74 di tumore ipofisario, 115 di meningioma e 100 di glioma.

La differenza tra i diversi adenomi sta nella forma della massa tumorale e la posizione in cui questa si trova, proprio perchè l'origine del tumore è differente. Ad esempio il glioma si sviluppa nelle cellule della glia e può insorgere in uno qualsiasi dei due emisferi cerebrali, mentre il meningioma cresce quasi sempre intorno alla parete meningea ed ossea. Il tumore ipofisario prende origine dalla adenoipofisi e dunque è localizzato più internamente nell'encefalo rispetto agli altri.

Dunque il modello deve essere in grado di riconoscere e classificare l'immagine esclusivamente lavorando su queste caratteristiche.

Una classificazione di questo tipo è detta *multiclasse*.

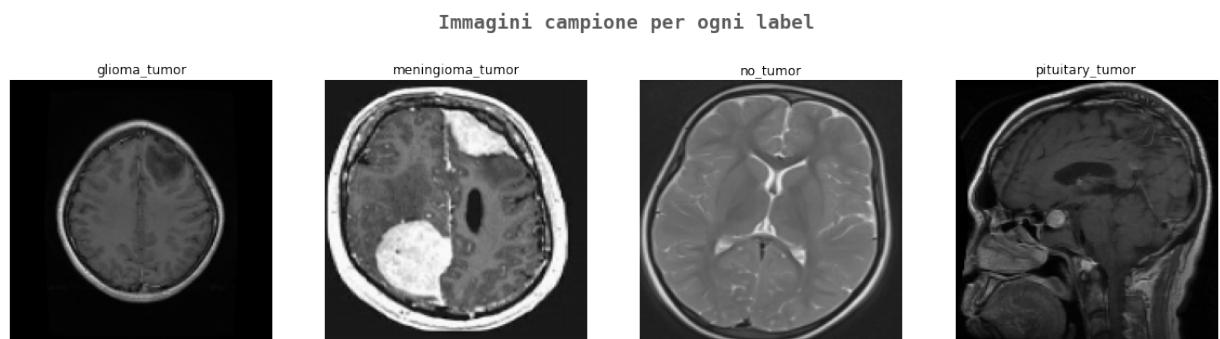


Figura 5.7: Campione di risonanza magnetica per ognuna delle diagnosi. (Disponibile codice in sezione 7)

Non si conoscono dettagli sui pazienti ai quali è stata fatta la risonanza. Inoltre, in questo caso il dataset è abbastanza povero per poter andare a classificare 4 categorie di patologie differenti. Proprio per questo è stato utilizzato un modello con più strati e varie tecniche per ampliare il dataset .

5.3.2 Setup iniziale

Come nell'esperienza precedentemente illustrata, anche in questo caso la prima cosa da fare è importare le librerie necessarie per la creazione del modello e anche in questo caso le classi importate sono le stesse della sezione 5.2.2. In più però sono stati inseriti nei modelli utilizzati degli strati aggiuntivi, in particolare i due che seguono.

- Dropout Layer: [28] come accennato sopra, il dimensionamento di una Rete Neurale è un compito compless. Una Rete Neurale troppo piccola potrebbe non essere in grado di riconoscere abbastanza tratti per classificare efficacemente gli ingressi, mentre una troppo grande potrebbe ottenere prestazioni estremamente buone in fase di addestramento ma ottenerne di pessime in fase di test, poichè durante l'addestramento ha sviluppato delle attivazioni nascoste che le permettevano di ottenere un maggior livello di accuracy sul training. È buona norma dimensionare una rete leggermente in eccesso (rispetto a quanto si pensa sia ottimale o a quanto risulta ottimale) per poi utilizzare il Dropout, metodo per evitare la creazione delle attivazioni nascoste. Lo strato di Dropout infatti fa sì che le unità che riceve in input un determinato strato della rete vengano randomicamente settate a 0 con una frequenza che deve essere inserita come parametro. Essa indica la probabilità che un neurone si sconnetta dai suoi precedenti. Il Dropout fa sì che la rete impari informazioni più robuste proprio perché viene ridotto in maniera casuale in numero di connessioni, così che i nodi rimasti connessi dovranno regolare i propri pesi per adattarsi all'assenza dei nodi non connessi, scalando i pesi di un valore tale che la somma pesata degli input sia sempre la stessa. Ciò permette di ridurre overfitting, se posizionati nei punti giusti. Di solito uno strato di Dropout viene posizionato con un rate di 0.5 tra uno strato denso e l'altro della CNN.

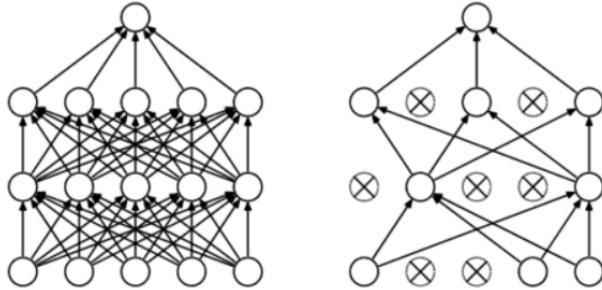


Figura 5.8: Dimostrazione schematica di cosa fa il dropout [28]. La prima immagine rappresenta una rete neurale connessa senza dropout, mentre la seconda rappresenta la rete dopo il dropout.

- BatchNormalization Layer [29]: serve anche questo a dare maggior robustezza alla rete. Le ragioni per le quali essa funziona sono ancora sotto discussione. Si crede che questo possa mitigare il problema dello *spostamento della covarianza interna*, secondo il quale l'inizializzazione di parametri e i cambiamenti nella distribuzione degli input, portano gli strati nascosti a doversi adattare ai nuovi, rallentando il training della rete. Altri studiosi pensano che invece tale strato porti al problema dell'esplosione del gradiente, contrario di quello di cui si discuteva nella sezione 3.2.3. Altri ancora negano ciò che è stato detto sopra, affermando che tale strato semplicemente serve a fare uno *smoothing* della funzione di errore. Sta di fatto che di solito se posizionati subito dopo gli strati di convoluzione in una CNN, rappresentano delle tecniche euristiche utili per il miglioramento del training.

5.3.3 Definizione degli iperparametri e dei modelli utilizzati

In questa esperienza sono stati utilizzati 3 modelli differenti e per ognuno di essi sono stati testati vari iperparametri.

All'inizio è stato utilizzato un modello molto simile a quello della sezione 5.2.4 ma con più strati e filtri in quanto è stato pensato che l'estrazione di una maggior quantità di features potesse essere utile per sopperire alla grandezza limitata del dataset a disposizione.

Successivamente è stato utilizzato un modello con architettura nota ovvero **AlexNet** [30]⁴, la prima implementazione di una CNN basata sull'utilizzo della GPU per accelerare il training ad avere vinto un contest di riconoscimento di immagine. Vi sono diverse versioni di tale rete e quella utilizzata in tale esperienza è stata riadattata rispetto all'originale in modo da soddisfare gli obiettivi preposti per la classificazione.

Infine è stato utilizzato un modello pre-addestrato di una rete chiamata EfficientNetB0 [31]⁵ tramite una tecnica chiamata *Transfer Learning*, la quale si basa sull'utilizzo di reti precedentemente allenate su un dataset differente. In effetti i primi strati di una CNN molto profonda in genere contengono estrattori di feature più generiche (esempio rilevatori di bordi o di blob di colore) mentre quelli finali sono più specifici per una singola classificazione. Pertanto andando a utilizzare i pesi inizializzati della rete preformata e andando a riqualificare gli ultimi strati a seconda del caso di interesse, è possibile che si riescano ad estrarre le feature anche dai nuovi dati.

Sotto si riportano dunque le architetture dei 3 modelli utilizzati:

1. Una CNN generica con 4 strati di convoluzione 2D (sono stati utilizzati 32, 32, 64 e 128 filtri per strato) ognuno seguito da una BatchNormalization, disposti in due blocchi alla fine dei quali sono stati aggiunti due strati di MaxPooling-Dropout.
I Fully Connected Layers sono 3, uno con 512, l'altro con 128 e l'ultimo con 4, ognuno intervallato da una BatchNormalization.
Le funzioni di attivazione sono sempre ReLu per tutti gli strati nascosti e l'ottimizzatore scelto è Adam (sarà così anche per i modelli successivi) in quanto risultano essere i migliori per quanto riguarda un buon compromesso tra durata del training e accuratezza della rete.

⁴Il nome di questa archittura viene da Alex Krizhevsky, un computer scientist ucraino, che con tale CNN ha vinto l'ImageNet Challenge nel 2012. Il progetto ImageNet è un ampio database, nato per la ricerca software nel campo della *visual object recognition*.

⁵EfficientNet si basa sullo studio dello *scaling* delle reti. Scalare una rete è qualcosa che viene fatto per incrementare l'accuracy di una CNN. Fare scaling di una CNN significa ridimensionare questa, e ciò può essere fatto in profondità (andando a aumentare il numero di strati di una CNN), in larghezza (andando a regolare il numero di filtri) e per risoluzione (andando a mettere in input immagini più o meno rumorose). È necessario andare a bilanciare questi 3 fattori tra loro e gli autori di questa rete hanno fatto vari esperimenti andando a utilizzare diversi valori di scaling fino a proporre un coefficiente composto che andasse a definire un buon legame tra di essi, tirando fuori una nuova famiglia di modelli che hanno raggiunto un'efficienza e un'accuracy molto alte (ad oggi 84.3% per ImageNet)

SEZIONE 5.3. CNN PER LA CLASSIFICAZIONE DI RISONANZE MAGNETICHE CEREBRALI

La funzione di perdita utilizzata nella classificazione multipla è la `categorical_crossentropy`, la quale permette la realizzazione dell'algoritmo di backpropagation calcolando l'errore di previsione tra il vettore rappresentante la label di target (infatti in questo caso la coppia immagine-label presenta un numero di label codificato in codice one-hot⁶) e il vettore di stima elaborato fino a quel momento. Per i vari training eseguiti sono stati utilizzati questi iperparametri:

- Una `image_size = 150` dove si intende che tutte le immagini in input alla rete sono state, una volta lette, ridimensionate con un numero di pixel pari a `image_size x image_size`.
A questa è stata associato un valore di `batch_size` che è stato fatto variare tra 8, 32 e 64. Successivamente è stata utilizzata anche una `image_size` di 224;
- Il numero di epoche è stato posto inizialmente per tutti i modelli pari a 40 ma poi il numero è stato ridotto in quanto si è notato che il training risultava essere stagnante su un minimo già molto prima della 40-esima epoca;
- Il valore di `hyper_featuremaps` è stato posto pari a 32 e mano a mano che si giunge più in profondità il numero dei filtri raddoppia (64 filtri nel terzo strato) e quadruplica (128 filtri nel quarto), come è possibile vedere nel code listing sotto;
- `hyper_channels = 3`, ma è stato tentato un training anche con tale valore pari a 1.

⁶tipo di codifica in cui viene rappresentato un numero intero positivo attraverso un vettore che contiene solamente zeri eccetto la cella puntata dall'indice corrispondente all'intero positivo che si vuole codificare.

```
model = Sequential()
model.add(Conv2D(hyper_featuremaps, kernel_size=(3, 3),
                activation='relu',
                input_shape=(image_size,
                            image_size, hyper_channels)))
)
model.add(BatchNormalization())

model.add(Conv2D(hyper_featuremaps , kernel_size=(3, 3),
                activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(hyper_featuremaps*2, kernel_size=(3, 3),
                activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.25))

model.add(Conv2D(hyper_featuremaps*4, kernel_size=(3, 3),
                activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())

model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(4, activation='softmax'))

model.compile(loss=keras.losses.categorical_crossentropy ,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Code Listing 5.3: Codice Python del primo modello utilizzato.

SEZIONE 5.3. CNN PER LA CLASSIFICAZIONE DI RISONANZE MAGNETICHE CEREBRALI

2. AlexNet [30]s nella sua prima versione del 2012 era costituito da 8 strati di cui 5 di convoluzione e 3 fully-connected da 4096, 4096 e 1000 unità in quanto tale rete è nata per una classificazione per 1000 classi diverse. AlexNet è nato infatti per un dataset di larga scala come ImageNet, con più di 15 milioni di immagini ad alta risoluzione appartenenti a più di 22.000 categorie differenti. Per mantenere l'architettura il più simile possibile a quella originale tutti gli strati sono stati mantenuti, ma ne è stato aggiunto un nono per andare a classificare le 4 tipologie di diagnosi, dunque di 4 unità. Il primo strato di convoluzione filtra un'immagine con `image_size=150` e `hyper_channels = 3` con uno stride di 4 pixel. Il secondo strato convoluzionale prende in input l'output (normalizzato e passato per la pool) del primo strato di convoluzione con 256 filtri di dimensione 5×5 . Da qui in poi lo stride per la convoluzione è sempre pari ad 1, mentre è pari a 2 quello per il pooling. Il terzo, il quarto e il quinto strato di convoluzione sono connessi tra loro senza essere separati da uno strato di pooling. Il terzo strato di convoluzione ha 384 kernel di dimensione 3×3 e così il quarto, mentre il quinto utilizza 256 filtri ed è seguito da uno strato di Max-Pooling.

È stata scelta questa rete perchè non troppo complessa tra le reti più conosciute e facile da manipolare e riutilizzare anche per altri tipi di classificazione.

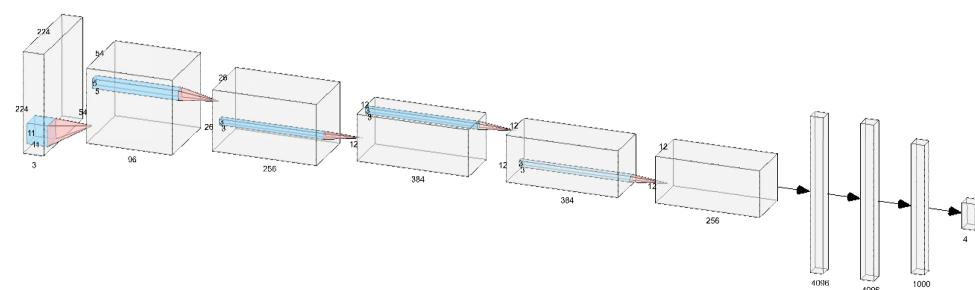


Figura 5.9: schema di Alexnet riadattato, realizzato con NNSVG.

```
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu',
                        input_shape=(image_size, image_size,3)),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu',
                        padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu',
                        padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu',
                        padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu',
                        padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1000, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4, activation='softmax')])
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

Code Listing 5.4: Codice Python del modello di AlexNet riadattato.

3. Nella terza fase di questa esperienza si applica il concetto di Transfer Learning. Viene infatti utilizzato un modello già allenato per il dataset ImageNet, ovvero EfficientNetB0. La funzione omonima EfficientNetB0() ritorna un modello di Keras di questo tipo.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels \hat{C}_i	#Layers \hat{L}_i
1	Conv3x3	224×224	32	1
2	MBCConv1, k3x3	112×112	16	1
3	MBCConv6, k3x3	112×112	24	2
4	MBCConv6, k5x5	56×56	40	2
5	MBCConv6, k3x3	28×28	80	3
6	MBCConv6, k5x5	28×28	112	3
7	MBCConv6, k5x5	14×14	192	4
8	MBCConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

Figura 5.10: Immagine presa dal paper di EfficientNet. Essa mostra come è fatta l'architettura di EfficientNetB0. Ogni riga descrive un passaggio i con un numero L_i di strati, con risoluzione in input pari a $H_i \times W_i$ e numero di filtri in output pari a C_i . Si sfrutta uno strato di convoluzione seguito da 7 blocchi di MBCConv (vedasi sotto) e che termina con uno strato FC.

Un blocco *MBCConv* Consiste usa un approccio *narrow-wide-narrow* [32] con una convoluzione 2D 1x1 seguita da una convoluzione depthwise che permette di ridurre il costo computazionale del training e che riduce il numero di parametri. Dopo di che, un altro strato di convoluzione fa uno squeezing della rete così che il numero di filtri coincide con quello desiderato.

Con la convoluzione Depthwise si può vedere come effettivamente da 80 milioni di parametri si passa a solo 4 milioni circa se gli iperparametri sono quelli iscritti nei code listings sopra.

```
def MBCConv1(x, expand=expansion_number, squeeze=squeeze_number):
    m = Conv2D(expand, (1,1))(x)
    m = BatchNormalization()(m)
    m = Activation('relu6')(m)
    m = DepthwiseConv2D((3,3))(m)
```

```
m = BatchNormalization()(m)
m = Activation('relu6')(m)
m = Conv2D(squeeze, (1,1))(m)
m = BatchNormalization()(m)
return Add()([m, x])
```

La funzione EfficientNetB0 di TensorFlow dà la possibilità di settare il parametro `include_top = False`, così che gli strati densi del modello originale non vengano importati in modo da riadattare la sua architettura agli obiettivi di questo lavoro.

In effetti gli strati densi del modello e l'operazione di Flattening sono stati successivamente sostituiti dall'operazione di GlobalAveragePooling2D, equivalente al MaxPooling ma che utilizza il valore medio per ogni feature map anzichè il massimo e che utilizza una pool di dimensione pari alle dimensioni delle mappe di input. Successivamente, si aggiunge un Dropout e infine uno strato denso che consente la classificazione delle 4 categorie.

```
effnet = EfficientNetB0(weights='imagenet', include_top=False,
                        input_shape=(image_size,
                                     image_size, hyper_channels))

model = effnet.output
model = tf.keras.layers.GlobalAveragePooling2D()(model)
model = tf.keras.layers.Dropout(rate=0.5)(model)
model = tf.keras.layers.Dense(4, activation='softmax')(model)
model = tf.keras.models.Model(inputs=effnet.input, outputs =
                               model)

model.summary()
model.compile(loss='categorical_crossentropy', optimizer =
               Adam', metrics= ['accuracy']
)
```

Code Listing 5.5: Codice realizzato per applicare il Trasfer Learning.

I 3 modelli hanno in comune lo stesso ottimizzatore cioè Adam e la stessa funzione di perdita, così come le funzioni di attivazione utilizzate (eccetto EfficientNet che utilizza talvolta la *swish* al posto della ReLu), in modo particolare quella dell'ultimo

stato che è la *softmax*. Questa è infatti utilizzata di prassi nella classificazione multipla. Essa comprime un vettore z k-dimensionale a valori reali in un vettore k-dimensionale $\sigma(z)$ di valori compresi nell'intervallo (0,1) la cui somma è 1.

Applicare la softmax significa prendere in considerazione tutti gli elementi dell'output dell'ultimo strato e definire da questi una funzione di probabilità considerandoli correlati tra loro. In effetti la funzione è di questo tipo:

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

dove K è il numero di classi. Dunque nel caso di interesse in output alla rete di otterrà un vettore 4x1 ove ogni riga k rappresenta la distribuzione di probabilità per cui l'input dato in impasto alla rete possa appartenere alla k-esima classe. Ovviamente tale funzione è strettamente legata all'uso della funzione di entropia incrociata categorica.

5.3.4 Creazione dei set di training e testing

Dopo aver letto (utilizzando la libreria openCV) e ridimensionato tutte le immagini, queste vengono inserite in un array \mathbf{X} il quale viene convertito poi in un NumPy array in cui ogni elemento è associato all'array di label \mathbf{y} corrispondenti tra le 4 possibili, come prevede l'apprendimento supervisionato.

```
labels = ['glioma_tumor', 'meningioma_tumor', 'no_tumor', 'pituitary_tumor']
```

Questa volta per questo tipo di dataset è stato deciso di utilizzare la funzione della libreria scikit ovvero `train_test_split`, che permette di suddividere la coppia (\mathbf{X}, \mathbf{y}) in due set di training e testing, dando la possibilità di decidere le dimensioni dei due rispetto al totale degli elementi e di fare uno shuffle a questi, così che l'apprendimento sia più robusto (ovviamente ogni volta che si richiama tale funzione si ottengono set di testing e training diversi, pertanto i risultati ottenuti possono variare leggermente gli uni dagli altri) È stato scelto un set di testing pari al 10% del totale, dunque 327 elementi contro i 2894 del training, poichè si è preferito lasciare il massimo numero di immagini possibili riservate all'apprendimento. Ovviamente a seconda di come vengono disposte le

immagini nel set di training e in quello di testing si possono ottenere risultati differenti per quanto riguarda l'accuratezza della rete, pertanto in media sono stati fatti 5 training per ogni modello. In questo caso non avendo a disposizione un dataset sufficientemente fornito, cosa abbastanza tipica dei dataset biomedicali, è stato deciso di far coincidere set di validazione e set di testing. A questo punto si è operato in maniera differente per ognuno dei 3 modelli descritti alla sezione precedente.

5.3.5 Fase di fitting e predizioni

1. Con il primo modello il numero di parametri da allenare è piuttosto alto (75,932,388 parametri se l'immagine di ingresso ha `image_size = 150`) causa il grande numero di filtri applicati e di unità negli strati densi. Inizialmente è stato fatto un aumento del numero di immagini con tecniche simili a quelle applicate nel dataset precedente ma ciò non ha portato a miglioramenti rispetto all'accuratezza sull'insieme di test.

Questo perchè in questo caso ciò che comporta una maggiore accuratezza alla rete è la capacità di distinguere la forma di un meningioma da quella di un glioma e un adenoma ipofisario anche in rapporto alla posizione in cui si trovano all'interno dell'encefalo. Un modello che possiede molti parametri da allenare con l'aggiunta di immagini ruotate e alterate geometricamente ha portato infatti ad un lieve overfitting, passando da un'accuratezza sul testing media di 92.2% (valore medio calcolato su 5 diversi training) a una media del 91%.

Dunque si è preferito allenare questo primo modello con il set autentico e ciò ha portato a risultati abbastanza soddisfacenti. I parametri che hanno portato ad un'accuratezza sull'insieme di test del 91.98% sono questi:

`image_size = 150` e `batch_size = 32`; un aumento ulteriore della grandezza dell'immagine portava ad overfitting.

`epochs=40`, di cui in media ne sono necessarie solo 20, ognuna delle quali con la GPU aveva la durata di circa 6 secondi, a fronte dei 3 minuti e 30 secondi a epoca con la necessità di almeno 20 epoche se si utilizza la CPU.

`hyper_channels = 3`, ovvero è stata mantenuta la modalità `rgb` con cui la libreria OpenCV legge le immagini. È stata fatta una prova anche con 1 solo canale ma non si sono ottenuti cambiamenti sostanziali.

SEZIONE 5.3. CNN PER LA CLASSIFICAZIONE DI RISONANZE MAGNETICHE CEREBRALI

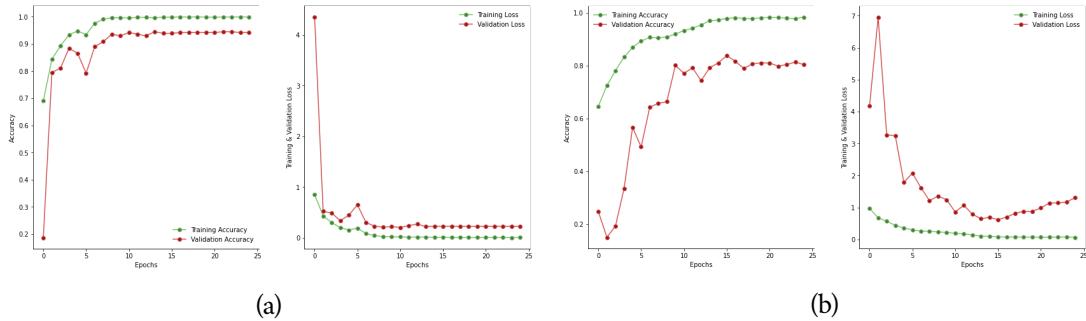


Figura 5.11: Esempi di andamento della validation e training accuracy-loss nel training del modello 1. A sinistra il training fatto sul set originale, a destra quello fatto sul set alterato. Come si vede nella figura (b) il set alterato non comporta miglioramenti in fase di testing e con le epoche cresce anche l'errore di stima.

2. Con la versione di AlexNet riadattata al caso di interesse i risultati ottenuti senza ampliare il dataset sono stati un'accuratezza sul training del 98% e sul testing del $(92.6 \pm 1)\%$. Anche in questo caso, come per il modello precedente, andare a ad applicare tecniche di ampliamento del dataset che alterassero la geometria delle immagini non è stato molto utile e così nemmeno andare a traslare o ingrandire l'immagine, questo perchè la caratteristica principale delle CNN è di assicurare l'estrazione di feature che siano invarianti rispetto alle traslazioni dell'immagine. Invece tutte quelle tecniche che non alterano geometricamente la forma dell'immagine ma solamente l'intensità dei pixel, sia localmente sia in tutta l'immagine, possono aiutare sicuramente nel riconoscimento di determinate forme. Infatti le immagini biomedicali, in particolare quelle di questa tipologia, essendo acquisite con diversi scanner e in differenti locazioni, possono essere intrinsecamente molto eterogenee nell'intensità dei pixel, e nella saturazione. Quindi vi possono essere immagini più o meno definite e andare a regolare l'intensità dei pixel per tutte le immagini garantendo una maggiore omogeneità tra queste è sicuramente la via migliore per incrementare l'accuratezza della rete [33]. In particolare è stata applicata una perturbazione data da *rumore Gaussiano* (di valore medio nullo) al set di immagini per il training, facendo variare il valore della deviazione standard e accostando l'aggiunta di tale rumore ad un rescaling. Risultati buoni sono stati ottenuti con varianza pari a 2.

Qui sotto dei campioni di immagini ottenute sommando al Numpy array `X_test` ottenuto dal set di training originale il valore

```
gaussian = np.random.normal(mean, std, (image_size,
                                         image_size,
                                         hyper_channels))
```

dove `mean` è il valore medio e `std` la varianza del segnale gaussiano generante il rumore.

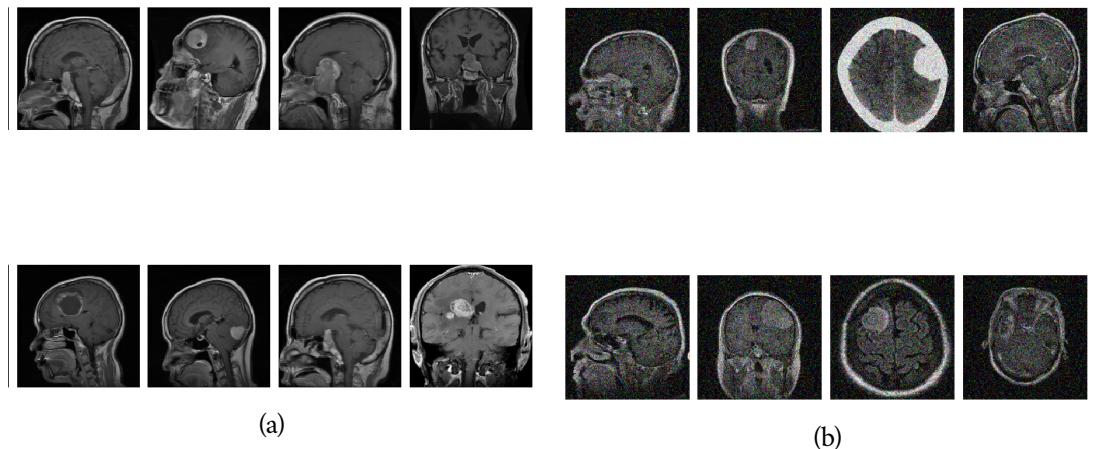


Figura 5.12: Esempi di RM prima di aver applicato il rumore gaussiano (a) e dopo (b). In questo caso la varianza è stata posta pari a 20 volutamente esagerando per dare maggior idea di che cosa comporta, ma in generale si utilizza un valore più basso per non alterare troppo le immagini.

Successivamente è stato applicato anche un `rescaling = 1./255` tramite `ImageDataGenerator`. In questo modo è stata raggiunta un'accuratezza sul set di testing pari al $(95.7 \pm 1)\%$, circa il $(2\text{-}3)\%$ in più rispetto al caso senza noising. Molti studi hanno notato infatti che aggiungere piccole quantità di rumore (*jitter*) nel set di training aiuta nella generalizzazione e per la *fault tolerance*, cioè la capacità che ha la rete di continuare ad essere affidabile anche se alcune delle sue unità nascoste smettono di funzionare. Aggiungere rumore alle immagini significa che la rete è meno capace di memorizzare i campioni di training perché questi cambiano continuamente così come le feature estratte, ottenendo valori dei pesi tra i neuroni

SEZIONE 5.3. CNN PER LA CLASSIFICAZIONE DI RISONANZE MAGNETICHE CEREBRALI

più piccoli ma una maggior robustezza. Infatti se una rete viene allenata con immagini affette da rumore sufficientemente piccolo, poi avrà maggior capacità di riconoscere i pattern nelle immagini di testing non affette da tale errore. In questo caso l'accuratezza nel training è circa del $(97.5 \pm 1)\%$, quindi inferiore a quella del modello in cui non è stato aggiunto il rumore, ma a discapito di una migliore performance in fase di testing. Inoltre è stato approvato che a parità di epoche il training con il rumore aggiunto raggiunge risultati migliori a livello di validation accuracy, dunque il sistema riesce ad allenarsi più velocemente.

Gli iperparametri scelti sono gli stessi del modello precedente.

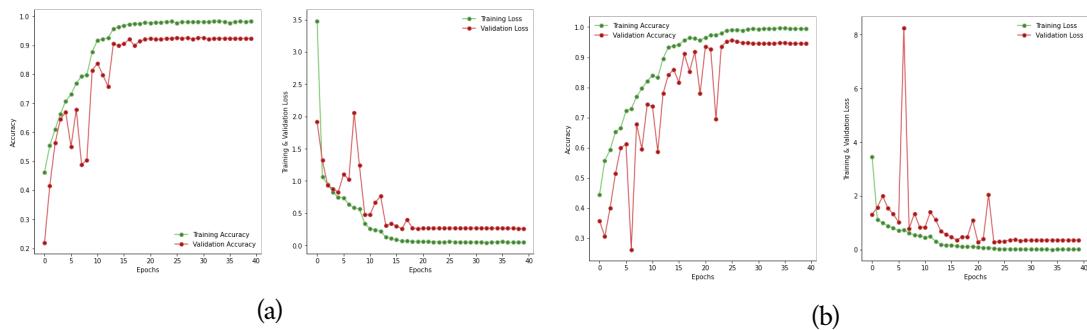


Figura 5.13: Esempi di andamento della validation e training accuracy-loss del secondo modello. A sinistra il training fatto sul set originale, a destra quello fatto sul set alterato con il rumore gaussiano.

3. Dato che i modelli precedenti, così detti "*from scratch*", hanno raggiunto risultati non superiori al $(95.7 \pm 1)\%$, è stato eseguito il training sul modello di EfficientNet descritto al punto 3 della sezione precedente. In questo caso in 14 epocha è stata raggiunta un'accuratezza sia sul training sia sul testing in media del $(99 \pm 0.1)\%$. Infatti per le tecniche sopracitate di tipo depth-wise utilizzate nell'architettura del modello il numero di parametri, sempre con `image_size = 150`, `batch_size = 32` e `hyper_channels=3` si ottengono solo 4,054,695 parametri, di cui circa 42 000 di tipo non-trainable.

In dataset piccoli come questo infatti il concetto di transfer-learning basato su modelli allenati sull'ImageNet rappresenta lo stato dell'arte per i problemi di classificazione e segmentazione di immagine.

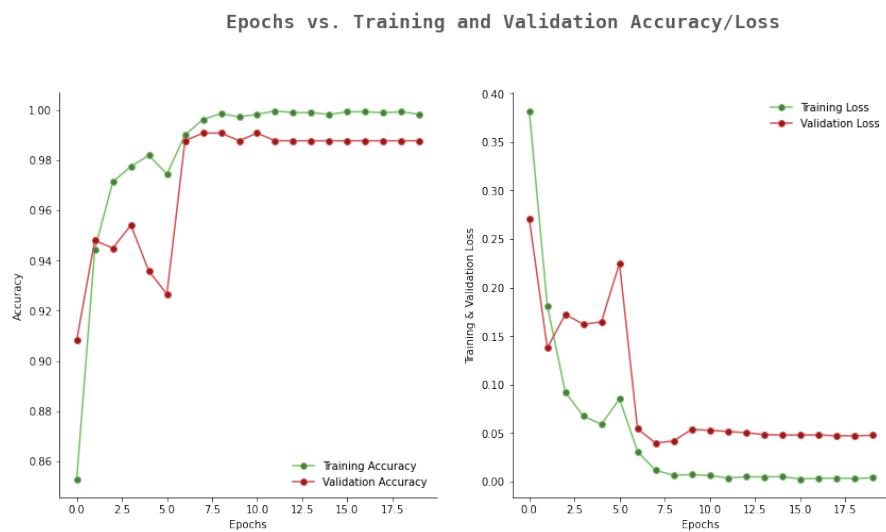


Figura 5.14: Andamento della training-validation accuracy e loss del modello 3.

Si riportano le tabelle riferite alla capacità di predizione della rete in seguito all'utilizzo dei 3 modelli e le matrici di confusione corrispondenti.

SEZIONE 5.3. CNN PER LA CLASSIFICAZIONE DI RISONANZE MAGNETICHE CEREBRALI

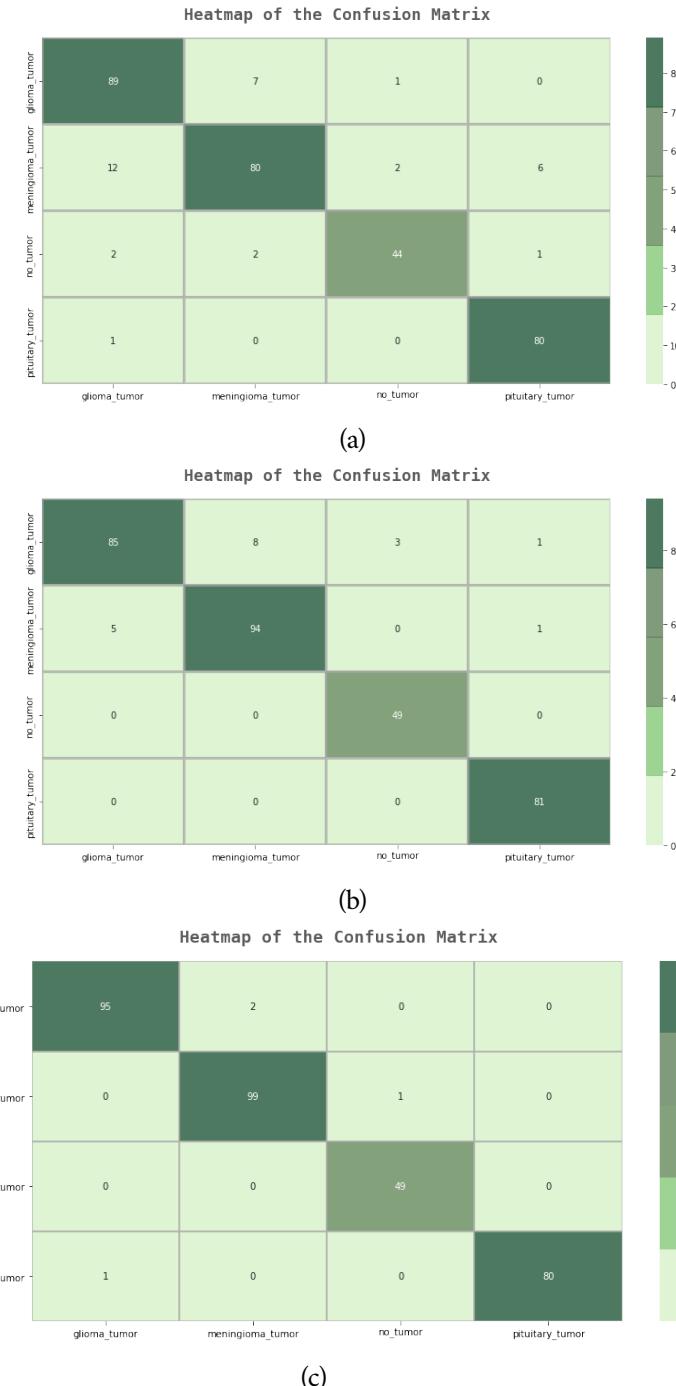


Figura 5.15: 3 matrici di confusione del primo modello (a), di AlexNet (b) (set a cui è stato applicato il rumore) e di Efficient-Net (c) pre-addestrato. Nella diagonale superiore vi sono i falsi positivi e in quella inferiore i falsi negativi. Si può notare che i falsi negativi diminuiscono a mano a mano che il sistema è più accurato.

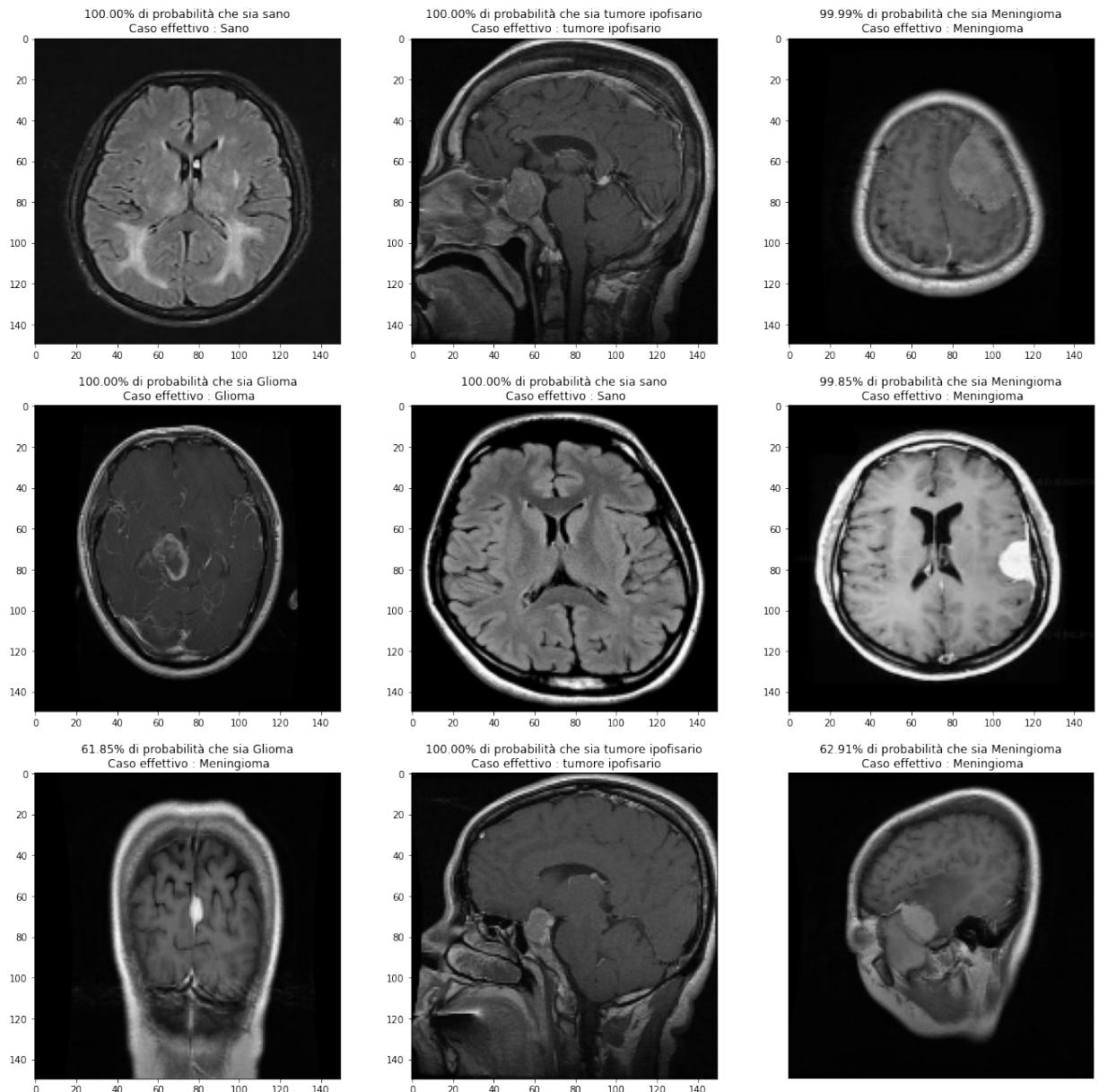


Figura 5.16: Predizioni fatte dal modello con associata la relativa probabilità versus quelle effettive su un insieme di immagini (codice di realizzazione nel capitolo 7).

5.3.6 Considerazioni finali

In tutti e 3 i casi si sono raggiunti risultati soddisfacenti, nonostante il dataset portasse con sè non poche difficoltà per il suo numero limitato di immagini. Sicuramente a differenza del dataset della polmonite, in cui la classificazione era binaria e ciononostante erano a disposizione un maggior numero di esempi, in questi casi in cui la classificazione è multiclassa la scelta migliore è andare a utilizzare un modello preesistente.

Se si vogliono ottenere buoni risultati partendo da 0 è necessario un buon modello con un numero di parametri da allenare che non sia troppo elevato (come nel caso del primo modello) se non si vuole impiegare troppo tempo nel training rispetto a quella che sono le effettive dimensioni dei dati e i risultati ottenuti, e che la scelta di come andare ad aggiungere rumore sia ben ponderata. Sono state fatte prove anche con `image_size` pari a 224, perchè tale numero di pixel per le immagini di input era risultato ottimale per AlexNet (vedasi paper del 2012 scritto dagli autori [30]), e con `batch_size` pari a 8, 32 e 64.

Il valore del learning rate è stato regolato come nel caso della polmonite automaticamente grazie all'ottimizzatore Adam, che parte da un valore di 0.01 e tramte la funzione di riduzione del learning rate, si riescono a superare i momenti di stagnazione dell'apprendimento uscendo al di fuori dei minimi locali della funzione di perdita. Si riportano a titolo di esempio i risultati medi ottenuti per 5 training del primo modello al variare degli iperparametri, i quali hanno portato a definire quelli ottimali e di fare fine-tuning (stessa cosa è stata ripetuta con le altre architetture).

	training accuracy	validation accuracy
batch = 8 image size = 150x150 channels = 3	99.9%	93.3%
batch = 32 image size = 150x150 channels = 3	99.9%	93.4%
batch = 64 image size = 150 x 150 channels = 3	98.9%	80.7%
batch = 32 image size = 150 x 150 channels = 1	99.7%	91.3 %
batch = 8 image size = 224x224 channels = 3	99.9 %	93.1%
batch = 32 image size = 224x224 channels = 3	99.8%	93.2%
batch = 64 image size = 224 x 224 channels = 3	99.9%	93.1%
batch = 32 image size = 224 x 224 channels = 1	99.8%	92.7%

Tabella 5.4: Si noti che tra tutti i training quello in cui si sono ottenuti i risultati in media migliori per il test è quello con gli iperparametri pari a quelli della seconda riga.

6. Conclusioni

Al termine delle due esperienze è stato possibile andare a classificare immagini biomedicali utilizzando reti neurali convoluzionali ed ottenere buoni risultati sia implementando e allenando la rete partendo da 0, sia utilizzando un modello precedentemente allenato.

È stato visto inoltre quanto sia necessario un buon dataset come punto di partenza così come vi debba essere un equilibrio tra le dimensioni del set di training e quello del set di testing, quelle degli iperparametri, facendo attenzione anche all'architettura della CNN scelta, così che non presenti troppi parametri rispetto alla complessità del problema di interesse, ma nemmeno che questi siano insufficienti.

La difficoltà principale nell'andare a realizzare sistemi come questi sta proprio nel dosare nella maniera più corretta tutti questi fattori, ed è necessario per riuscire compiere una quantità molto elevata di training, applicando la tecnica conosciuta come *trial and error*.

È stato visto come costruire tali modelli richieda diversi step, partendo da una corretta analisi del dataset, elaborare i primi risultati fino a trovare gli iperparametri ottimali in seguito a vari tentativi e procedere con il fine-tuning della rete di conseguenza, ampliando il dataset se necessario. Sono state passate in rassegna le tecniche per estendere il set di training di maggior uso fino a capire che nel caso di immagini biomedicali è bene non andare a caso nella scelta di queste e che tutto dipende da quelli che sono i pattern significativi da riconoscere.

È bene scegliere trasformazioni che non vadano a rivoluzionare troppo gli schemi di immagine veri e propri, dunque le trasformazioni geometriche possono essere utili solo se fatte nella misura giusta da non portare alla creazione di esempi anatomicamente

incorretti e ciò richiede sforzo nell'andare a ricercare quale siano le migliori. Andare ad aggiungere limitatamente trasformazioni *pixel-wise* nel caso del secondo dataset e delle leggere rotazioni e zoom alle immagini nel primo dataset sono state i modi in cui sono stati ottenuti i migliori risultati.

Ovviamente tutto questo funziona se il modello è ben strutturato, altrimenti questo potrebbe fare da collo di bottiglia al training.

Infine questa esperienza mostra come l'utilizzo delle CNN possa essere esteso a vari tipi di classificazione e che modelli come quelli illustrati possano essere utilizzati per altri tipi di dataset affini. È sicuramente possibile sfruttare i training fatti alle reti neurali sui due dataset e riutilizzarne i pesi per training successivi per l'identificazione di altre patologie, ad esempio andando a fare un'ulteriore distinzione tra polmonite virale o batterica in riferimento al modello ottenuto nella prima esperienza oppure per andare a riconoscere malattie neurologiche che comportano una mutazione anomala dell'encefalo (che quindi possono essere riconosciute tramite RM), come ad esempio quelle neurodegenerative (es. Alzehimer), usando il secondo sistema.

È possibile infatti durante ogni training salvare il modello allenato in un file e riutilizzarlo quando si vuole.

Si riportano adesso le curve *ROC* (*Receiver Operating Characteristics*) per mostrare la validità dei sistemi implementati. La curva ROC [34] è uno schema di rappresentazione grafica che permette di valutare l'efficienza di un classificatore binario. L'analisi della curva ROC è un metodo statistico utilizzato spesso in ambito biomedico. Essa rappresenta il metodo d'elezione per validare un test diagnostico. La curva ROC viene costruita mettendo in ascissa la *specificità* ($\frac{TN}{TN+FP}$) e in ordinata la *sensibilità* ($\frac{TP}{TP+FN}$, corrisponde a quella che nel capitolo 5 è stata definita come *recall*) per tutti i possibili valori del test diagnostico. L'area sotto la curva ROC, *area under curve* (AUC), è una misura quantitativa della bontà del classificatore, che può assumere valori tra 0.5 e 1.0. Se il valore della AUC è compreso tra 0,9 e 1.0 significa che il classificatore è altamente accurato e possiede alto potere discriminante. Tanto maggiore è l'area sotto la curva (cioè tanto più la curva si avvicina al vertice in alto a sinistra del grafico, quanto più somiglia ad un gradino) tanto maggiore è il potere discriminante della rete.

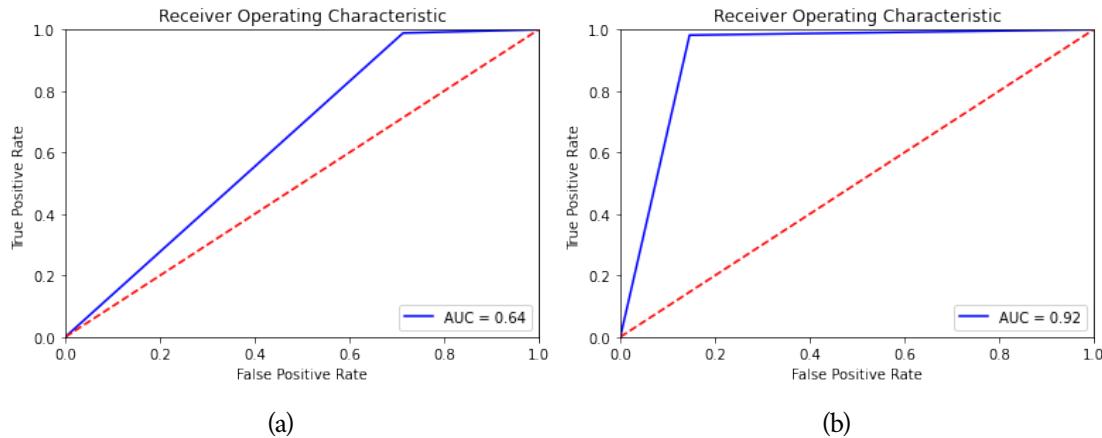


Figura 6.1: Confronto della caratteristica ROC tra il sistema per la rilevazione della polmonite senza applicare le modifiche al set di training (a) e quella del sistema allenato sul set ampliato (b). Il codice di realizzazione può essere visto nel Code Listing 7.1.

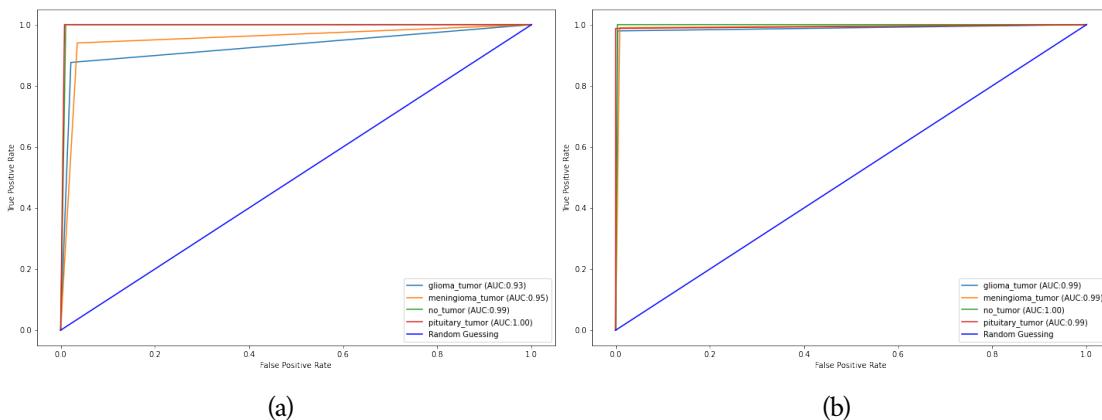


Figura 6.2: Confronto tra la caratteristica ROC del modello 2. con rumore gaussiano e del modello 3. per il dataset delle RM. (a) mostra una AUC di 0.96, (b) dello 0.99. Il codice di realizzazione può essere visto nel Code Listing 7.2.

7. Codice

Si riporta il codice Python utilizzato per il training del sistema per la classificazione dei raggi X e successivamente quello per la diagnostica delle RM.

Code Listing 7.1: Esempio di implementazione di una CNN per la classificazione di raggi-X. [25]

```
#Si importano le principali librerie
import matplotlib.pyplot as plt #per la visualizzazione
import numpy as np #per gestire gli array
import pandas as pd #per gestire i dati
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense,Conv2D,Flatten,MaxPooling2D
from tensorflow.keras.callbacks import EarlyStopping,ReduceLROnPlateau
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import classification_report,confusion_matrix
import sklearn.metrics as metrics #per fare le previsioni
import seaborn as sns
import scikitplot as skplt

#Si definiscono le directory per il training, testing e validation
train_path = './pneumonia/train'
test_path = './pneumonia/test'
valid_path = './pneumonia/val'

#Si definiscono gli iperparametri
batch_size = 16
img_height = 500
img_width = 500
channels = 1
hyper_epochs = 20

#Si applicano le tecniche di image preprocessing di cui si parla nella sezione 5.2.5 al set di training e testin
from tensorflow.keras.preprocessing.image import ImageDataGenerator# Create Image Data Generator for Train Set
image_gen = ImageDataGenerator(rescale = 1./255,
                               rotation_range = 5,
                               zoom_range = 0.2)
test_data_gen = ImageDataGenerator(rescale = 1./255)

train = image_gen.flow_from_directory(
    train_path,
    target_size=(img_height, img_width),
    color_mode='grayscale',
    class_mode='binary',
```

SEZIONE

```
batch_size=batch_size
)
test = test_data_gen.flow_from_directory(
    test_path,
    target_size=(img_height, img_width),
    color_mode='grayscale',
    shuffle=False,
    #si setta shuffle=False per non avere problemi di indicizzazione quando si va a fare il test
    class_mode='binary',
    batch_size=batch_size
)
valid = test_data_gen.flow_from_directory(
    valid_path,
    target_size=(img_height, img_width),
    color_mode='grayscale',
    class_mode='binary',
    batch_size=batch_size
)

#codice per stampare alcuni campioni del set di training che andranno in input alla rete
plt.figure(figsize=(12, 12))
for i in range(0, 10):
    plt.subplot(2, 5, i+1)
    for X_batch, Y_batch in train:
        image = X_batch[0]
        dic = {0:'NORMALE', 1:'POLMONITE'}
        plt.title(dic.get(Y_batch[0]))
        plt.axis('off')
        plt.imshow(np.squeeze(image),cmap='gray',interpolation='nearest')
        break
    plt.tight_layout()
plt.show()

#Si definisce il modello
cnn = Sequential()
cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(img_width, img_height, channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(img_width, img_height, channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(32, (3, 3), activation="relu", input_shape=(img_width, img_height, channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(64, (3, 3), activation="relu", input_shape=(img_width, img.height, channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Conv2D(64, (3, 3), activation="relu", input_shape=(img_width, img.height, channels)))
cnn.add(MaxPooling2D(pool_size = (2, 2)))
cnn.add(Flatten())
cnn.add(Dense(activation = 'relu', units = 128))
cnn.add(Dense(activation = 'relu', units = 64))
cnn.add(Dense(activation = 'sigmoid', units = 1))
#Si compila il modello applicando l'ottimizzatore e la funzione di errore
#sulla quale si basa l'algoritmo di backpropagation
cnn.compile(optimizer = 'adam', loss = 'binary_crossentropy', metrics = ['accuracy'])
cnn.summary()

#Si definiscono le callbacks che il sistema sfrutta nel training
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, TensorBoard, ModelCheckpoint
early = EarlyStopping(monitor= 'val_loss', mode='min', patience=5)
learning_rate_reduction = ReduceLROnPlateau(monitor='val_loss', patience = 2, verbose=1,factor=0.3,min_delta = 0.001, min_lr=0.000001)
checkpoint = ModelCheckpoint("my-model.h5",monitor="val_accuracy",save_best_only=True,mode="auto",verbose=1)
callbacks_list = [ early, learning_rate_reduction, checkpoint]

#Si vanno ad assegnare i valori dei pesi per ogni classe a seconda della cardinalita
#degli elementi presenti in essa.
#Dunque il modello assegna automaticamente i pesi della classe in maniera inversamente
#proporzionale alla frequenza degli elementi in una classe
weights = compute_class_weight('balanced', np.unique(train.classes), train.classes)
cw = dict(zip( np.unique(train.classes), weights))
```

CAPITOLO 7. CODICE

```
print(cw)

#Si fa il fitting
history = cnn.fit(train, epochs=hyper_epochs, validation_data=valid, class_weight=cw, callbacks=callbacks_list)

#Si crea un grafico dell'andamento
pd.DataFrame(cnn.history.history).plot()

#Si controlla l'accuratezza del test
test_accu = cnn.evaluate(test)
print('The_testing_accuracy_is_:',test_accu[1]*100, '%')
test_accu = cnn.evaluate(train)
print('The_training_accuracy_is_:',test_accu[1]*100, '%')
test.accu = cnn.evaluate(valid)
print('The_validation_accuracy_is_:',test.accu[1]*100, '%')

#Si fanno le previsioni
preds = cnn.predict(test, verbose=1)
predictions = preds.copy()
predictions[predictions <= 0.5] = 0
predictions[predictions > 0.5] = 1

#si genera la matrice di confusione
cm = pd.DataFrame(data=confusion_matrix(test.classes, predictions, labels=[0, 1]), index=["Sano_effettivo", "Polmonite_effettiva"], columns=["Previsto_sano", "Previsto_con_Polmonite"])
sns.heatmap(cm, annot=True, fmt="d")

#si stampa un report di classificazione
print(classification_report(y_true=test.classes, y_pred=predictions, target_names =['NORMALE', 'POLMONITE']))

#Si stampano alcuni esempi di immagini con associata la classe prevista e la probabilità con cui
#la previsione sia corretta, associata alla diagnosi effettiva (La figura 5.6 mostra l'output ottenuto)
test.reset()
x=np.concatenate([test.next()[0] for i in range(test.__len__())])
y=np.concatenate([test.next()[1] for i in range(test.__len__())])
print(x.shape)
print(y.shape)#this little code above extracts the images from test Data iterator without shuffling the sequence
# x contains image array and y has labels
dic = {0:'SANO', 1:'POLMONITE'}
plt.figure(figsize=(20,20))
for i in range(0+228, 9+228):
    plt.subplot(3, 3, (i-228)+1)
    if preds[i, 0] >= 0.5:
        out = ('{:.2%} di probabilità che sia Polmonite'.format(preds[i][0]))
    else:
        out = ('{:.2%} di probabilità che sia un soggetto sano'.format(1-preds[i][0]))
    plt.title(out+"\nCaso_effettivo: "+ dic.get(y[i]))
    plt.imshow(np.squeeze(x[i]))
    plt.axis('off')
plt.show()

data = []      # archivia tutte le batch di immagini prodotte nell'insieme di test
labels = []    # archivia le label associate alle immagini dell'array sopra
max_iter = 100 # massimo numero di iterazioni, a seconda della batch size e della dimensione del dataset
i = 0
for d, l in test:
    data.append(d)
    labels.append(l)
    i += 1
    if i == max_iter:
        break
data = np.array(data)
data = np.reshape(data, (data.shape[0]*data.shape[1],) + data.shape[2:])
```

SEZIONE

```
labels = np.array(labels)
labels = np.reshape(labels, (labels.shape[0]*labels.shape[1],) + labels.shape[2:])

# si calcolano i falsi positivi e i falsi negativi per produrre la caratteristica ROC
preds = cnn.predict(data)
preds[preds <= 0.5] = 0
preds[preds > 0.5] = 1

fpr, tpr, threshold = metrics.roc_curve(labels, preds)
roc_auc = metrics.auc(fpr, tpr)

plt.title('Receiver_Operating_Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC=%0.2f' % roc_auc)
plt.legend(loc = 'lower_right')
plt.plot([0, 1], [0, 1], 'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True_Positive_Rate')
plt.xlabel('False_Positive_Rate')
plt.show()
```

Code Listing 7.2: Esempio di implementazione di AlexNet per la classificazione di RM con l'aggiunta di rumore al set di training. [27]

```
#Si importano le librerie principali
from keras.callbacks import TensorBoard
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import cv2
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tqdm import tqdm
import os
from pathlib import Path
import itertools
from sklearn.utils import shuffle
from keras import Sequential
from sklearn.model_selection import train_test_split
from tensorflow.keras import EfficientNetB0
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau, TensorBoard, ModelCheckpoint
from sklearn.metrics import classification_report, confusion_matrix
import ipywidgets as widgets
import io
from keras.layers import Flatten, Dense, BatchNormalization, Activation, Dropout
from tensorflow.keras.utils import to_categorical
from PIL import Image
from IPython.display import display, clear_output
from warnings import filterwarnings
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, Input
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from keras.regularizers import l2
from tensorflow.keras import Model
import numpy as np
import cv2
import glob

#Si definiscono array di colori che vengono raggruppati in scuri, rossi e verdi
# e utilizzati in seguito per generare la matrice di confusione
colors_dark = ["#1F1F1F", "#313131", "#636363", "#AEAEAE", "#DADADA"]
```

CAPITOLO 7. CODICE

```
colors_red = ["#331313", "#582626", '#9E1717', '#D35151', '#E9B4B4']
colors_green = ['#01411C', '#4B6F44', '#4F7942', '#74C365', '#D0F0C0']

sns.palplot(colors_dark)
sns.palplot(colors_green)
sns.palplot(colors_red)

#si scrivono le labels sulle quale si vuol fare la classificazione
labels = ['glioma_tumor', 'meningioma_tumor', 'no_tumor', 'pituitary_tumor']

#Si iniziano a leggere tutte le immagini e ad aggiungerle tramite una lista
#e si convertono in un numpy array dopo averle rese tutte della stessa grandezza
X_train = []
y_train = []
batch_size = 32
image_size = 150
hyper_epochs = 40
for i in labels:
    folderPath = os.path.join('/content/brain-tumor-classification-mri', 'Training', i)
    for j in tqdm(os.listdir(folderPath)):
        img = cv2.imread(os.path.join(folderPath, j))
        img = cv2.resize(img, (image_size, image_size))
        X_train.append(img)
        y_train.append(i)
#opencv permette di leggere l'immagine e restituisce un numpy array
for i in labels:
    folderPath = os.path.join('/content/brain-tumor-classification-mri', 'Testing', i)
    for j in tqdm(os.listdir(folderPath)):
        img = cv2.imread(os.path.join(folderPath, j))
        img = cv2.resize(img, (image_size, image_size))
        X_train.append(img)
        y_train.append(i)
#si convertono in numpy array
X_train = np.array(X_train)
y_train = np.array(y_train)
X_train.shape

#Si suddivide il dataset in un set di training X_train e uno di testing X_test in maniera casuale
#e ad ogni immagine si associa la propria label y_train, y_test
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, train_size=0.9, random_state=101)

#Si mostra un campione per ogni label (l'output lo si puo vedere nella figura 5.7)
k=0
fig, ax = plt.subplots(1,4, figsize=(20,20))
fig.text(s='Immagini_campione_per_ogni_label', size=18, fontweight='bold',
          fontname='monospace', color=colors_dark[1], y=0.62, x=0.4, alpha=0.8)
for i in labels:
    j=0
    while True:
        if y_train[j]==i:
            ax[k].imshow(X_train[j])
            ax[k].set_title(y_train[j])
            ax[k].axis('off')
            k+=1
            break
    j+=1

#si aggiunge rumore gaussiano alle immagini del set di training
mean = 0
std = 2
gaussian = np.random.normal(mean, std, (image_size, image_size, 3))
X_train = X_train + gaussian
X_train = np.clip(X_train, 0, 255)

#come da prassi si fa un rescaling delle immagini
datagen = ImageDataGenerator(rescale = 1./255)
```

SEZIONE

```
val_data_gen = ImageDataGenerator(rescale = 1./255)
test_data_gen = ImageDataGenerator(rescale = 1./255)

#si convertono gli indici associati alle label in codice one-hot così che si possa
#applicare la funzione di entropia categorica come funzione di perdita
y_train_new = []
for i in y_train:
    y_train_new.append(labels.index(i))
y_train = y_train_new
y_train = tf.keras.utils.to_categorical(y_train)

y_test_new = []
for i in y_test:
    y_test_new.append(labels.index(i))
y_test = y_test_new
y_test = tf.keras.utils.to_categorical(y_test)

#Si definiscono le tuple con le coppie (X,y) per il training, testing e validation, dove in questo caso il set di testing
#coincide con quello di validation
train = datagen.flow(X_train, y_train, batch_size=batch_size)
test = test_data_gen.flow(X_test, y_test, batch_size=batch_size, shuffle=False)
valid = test_data_gen.flow(X_test, y_test, batch_size=batch_size)

#Si stampano le immagini affette dal rumore (vedi Figura 5.12)
plt.figure(figsize=(12, 12))
for i in range(0, 10):
    plt.subplot(2, 5, i+1)
    for X_batch, Y_batch in train:
        image = X_batch[0]
        dic = {0:'Glioma', 1:'Meningioma', 2:'No', 3:'Pituitary'}
        plt.title(dic.get(Y_batch[0][0]))
        plt.axis('off')
        plt.imshow(np.squeeze(image), cmap='gray', interpolation='nearest')
        break
    plt.tight_layout()
plt.show()

#Si definisce il modello utilizzato
model = keras.models.Sequential([
    keras.layers.Conv2D(filters=96, kernel_size=(11,11), strides=(4,4), activation='relu', input_shape=(image_size,image_size,3)),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding="same"),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPool2D(pool_size=(3,3), strides=(2,2)),
    keras.layers.Flatten(),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4096, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(1000, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(4, activation='softmax')
])
model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=keras.optimizers.Adam(),
              metrics=['accuracy'])
```

CAPITOLO 7. CODICE

```
model.summary()

#si salva il modello in un preciso file e si definisce la funzione che regola il learning rate
checkpoint = ModelCheckpoint("Model_for_brain_tumor_classification.h5",monitor="val_accuracy",save_best_only=True,mode="auto",verbose=1)
reduce_lr = ReduceLROnPlateau(monitor = 'val_accuracy', factor = 0.3, patience = 2, min_delta = 0.001,min_lr= 0.000001, mode='auto',verbose=1)

#si fa il training sul modello, utilizzando come set di training quello affetto da rumore
hist = model.fit(X_train,y_train, epochs=hyper_epochs, validation_data= (X_test,y_test), callbacks=[checkpoint,reduce_lr])

#Si mostra l'andamento del training per controllare se vi e stato overfitting underfitting
filterwarnings('ignore')

epochs = [i for i in range(hyper_epochs)]
fig, ax = plt.subplots(1,2, figsize=(14,7))
train_acc = hist.history['accuracy']
train_loss = hist.history['loss']
val_acc = hist.history['val_accuracy']
val_loss = hist.history['val_loss']

fig.text(s='Epochs_vs._Training_and_Validation_Accuracy/Loss',size=18,fontweight='bold',
         fontname='monospace',color=colors_dark[1],y=1,x=0.28,alpha=0.8)

sns.despine()
ax[0].plot(epochs, train_acc, marker='o',markerfacecolor=colors_green[2],color=colors_green[3],
           label = 'Training_Accuracy')
ax[0].plot(epochs, val_acc, marker='o',markerfacecolor=colors_red[2],color=colors_red[3],
           label = 'Validation_Accuracy')
ax[0].legend(frameon=False)
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('Accuracy')

sns.despine()
ax[1].plot(epochs, train_loss, marker='o',markerfacecolor=colors_green[2],color=colors_green[3],
           label = 'Training_Loss')
ax[1].plot(epochs, val_loss, marker='o',markerfacecolor=colors_red[2],color=colors_red[3],
           label = 'Validation_Loss')
ax[1].legend(frameon=False)
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Training & Validation Loss')

fig.show()

#Si controllano le previsioni sul set di testing e si fa un classification report
pred = model2.predict(test)
pred = np.argmax(pred, axis=1)
y_test_new = np.argmax(y_test, axis=1)
print(classification_report(y_test_new,pred,target_names =['GLIOMA','MENINGIOMA','NO_TUMOR', 'PITUITARY']))

#Si genera la matrice di confusione
fig,ax=plt.subplots(1,1, figsize=(14,7))
sns.heatmap(confusion_matrix(y_test_new,pred),ax=ax,xticklabels=labels, yticklabels=labels, annot=True,
            cmap=colors_green[::-1],alpha=0.7,linewidths=2, linecolor=colors_dark[3])
fig.text(s='Heatmap_of_the_Confusion_Matrix',size=18,fontweight='bold',
         fontname='monospace',color=colors_dark[1],y=0.92,x=0.28,alpha=0.8)

plt.show()

#Si genera la curva ROC andando a rappresentare la bonta della classificazione con la tecnica One vs. All
#cioe andando a controllare quanto la rete e in grado di afferrare completamente nuovi esempi per quella classe
#rispetto a tutte le altre senza commettere errori
import scikitplot as skplt
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, roc_auc_score
import matplotlib.pyplot as plt
```

SEZIONE

```
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import roc_curve, auc, roc_auc_score

fig, c_ax = plt.subplots(1,1, figsize = (12, 8))

# funzione per calcolare la curva roc per ogni classe
def multiclass_roc_auc_score(y_test, pred, average="macro"):
    lb = LabelBinarizer()
    lb.fit(y_test)
    y_test = lb.transform(y_test)
    y_pred = lb.transform(pred)

    for (idx, c_label) in enumerate(labels):
        fpr, tpr, thresholds = roc_curve(y_test[:,idx].astype(int), y_pred[:,idx])
        c_ax.plot(fpr, tpr, label = '%s (AUC:%.2f)' % (c_label, auc(fpr, tpr)))
    c_ax.plot(fpr, fpr, 'b-', label = 'Random_Guessing')
    return roc_auc_score(y_test, y_pred, average=average)

print('ROC_AUC_score:', multiclass_roc_auc_score(y_test, pred))

c_ax.legend()
c_ax.set_xlabel('False_Positive_Rate')
c_ax.set_ylabel('True_Positive_Rate')
plt.show()

# Si stampano alcuni campioni di immagine con la probabilità associata sulla
# previsione fatta per essi, confrontandola con la diagnosi effettiva.
pred1 = model.predict(test)
pred1 = pred1
pred2 = np.argmax(pred1, axis=1)
y_test_new2 = np.argmax(y_test, axis=1)
print(y_test.shape)
x=np.concatenate([test.next()[0] for i in range(test.__len__())])
y=np.concatenate([test.next()[1] for i in range(test.__len__())])
print(x.shape)
print(y.shape)#this little code above extracts the images from test Data iterator without shuffling the sequence
print(pred1)
print(pred2)
print(y_test_new2)
print(y[1])
# x contains image array and y has labels
dic = {0:'Glioma', 1:'Meningioma', 2:'Sano', 3: 'tumore_ipofisario' }
plt.figure(figsize=(20,20))
for i in range(0+120, 9+120):
    plt.subplot(3, 3, (i-120)+1)
    if pred2[i] == 0:
        out = ('{:2%} - probabilità che sia Glioma'.format(pred1[i][0]))

    elif pred2[i] == 1:
        out = ('{:2%} - probabilità che sia Meningioma'.format(pred1[i][1]))
    elif pred2[i] == 2:
        out = ('{:2%} - probabilità che sia sano'.format(pred1[i][2]))
    else:
        out = ('{:2%} - probabilità che sia tumore_ipofisario'.format(pred1[i][3]))

    plt.title(out+"\nCaso_effettivo_:" + dic.get(np.argmax(y[i])))
    plt.imshow(np.squeeze(x[i]))
    plt.axis('off')
plt.show()
```

8. Bibliografia

- [1] P. Shukla and R. Iriondo, “Neural Networks from Scratch with Python Code and Math in Detail— I,” Accessed: September, 2021. [<https://bit.ly/3CTHArw>].
- [2] R. Iriondo, “What is Machine Learning?,” Accessed: September, 2021. [<https://pub.towardsai.net/what-is-machine-learning\protect\discretionary{\char\hyphenchar\font{}{}}ml-b58162f97ec7>].
- [3] Russell, Stuart, Norvig, and Peter, *Artificial Intelligence: A Modern Approach* (2nd ed.). Prentice Hall, 2003.
- [4] Dr. Michael Garbade, “Regression versus classification,” Accessed: September, 2021. [<https://bit.ly/2Yd0oDt>].
- [5] Gianluca Amato, “Classificazione e predizione,” Accessed: September, 2021. [<https://www.sci.unich.it/~amato/teaching/old/datamining08/\lucidi/05-classificazione.pdf>].
- [6] GitHubCommunity, “Apprendimento supervisionato,” Accessed: September, 2021. [<https://tvm1.github.io/ml1819/store/book.pdf>].
- [7] B. Mehlig, *Machine learning with neural networks*. University of Gothenburg, Feb 2021.
- [8] Matthew Roos, “Deep Learning Neurons versus Biological Neurons,” Accessed: September, 2021. [<https://bit.ly/3kQ3iqd>].
- [9] Osservatori Digital Innovation, “Alla scoperta del Deep Learning: significato, esempi e applicazioni,” Accessed: September, 2021. [blog.osservatori.net].

SEZIONE

- [10] Bellman and R. Ernest, *Dynamic programming*. Princeton University Press. University of Gothenburg, 1957.
- [11] Y. Mahdid, “GitHub: machine learning,” Accessed: September, 2021. [<https://bit.ly/3kWKdTw>].
- [12] Luca Marzano, “Algoritmi di deep learning per l’analisi di dati scientifici,” Accessed: September, 2021. [<https://cdlfbari.cloud.ba.infn.it/wp-content/uploads/file-manager/CIF/Triennale/Tesi%20di%20laurea/16-17-MARZANO%20Luca.pdf>].
- [13] “Fitting del modello: Fitting e overfitting,” Accessed: September, 2021. [https://docs.aws.amazon.com/it_it/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html].
- [14] T. Mitchell, *Machine Learning*. McGraw Hill, 1997.
- [15] A. Bonner, “The Complete Beginner’s Guide to Deep Learning: Convolutional Neural Networks and Image Classification,” Accessed: September, 2021. [<https://towardsdatascience.com/wtf-is-image-classification-8e78a8235acb>].
- [16] G. Cannata, “Come le reti neurali vedono le cose ,” Accessed: September, 2021. [<https://it.quora.com/Come-le-reti-neurali-profonde-vedono-le-cose>].
- [17] S. Dey, K.-W. Huang, P. A. Beerel, and K. M. Chugg, “Characterizing Sparse Connectivity Patterns in Neural Networks ,” Accessed: September, 2021. [<https://arxiv.org/pdf/1711.02131.pdf>].
- [18] H. Wu and X. Gu, “Max-Pooling Dropout for Regularization of Convolutional Neural Networks ,” Accessed: September, 2021. [<https://arxiv.org/ftp/arxiv/papers/1512/1512.01400.pdf>].
- [19] C. F. Wang, “The Vanishing Gradient Problem,” Accessed: September, 2021. [<https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484>].

- [20] E. Allibhai, “Building a Convolutional Neural Network (CNN) in Keras ;” Accessed: September, 2021. [<https://towardsdatascience.com/building-a-convolutional-neural-network-cnn-in-keras-329fbbadc5f5>].
- [21] C. D. Costa, “Best Python Libraries for Every Python Developer ,” Accessed: September, 2021. [<https://towardsdatascience.com/best-python-libraries-for-every-python-developer-77daab4fa40e>].
- [22] “Offical keras documentation,” Accessed: September, 2021. [<https://keras.io/>].
- [23] “Tensorflow documentation,” Accessed: September, 2021. [<https://www.tensorflow.org/>].
- [24] “Jupyter documentation,” Accessed: September, 2021. [<https://jupyter.org/>].
- [25] P. Breviglieri, “Pneumonia X-Ray images ,” Accessed: September, 2021. [<https://www.kaggle.com/pcbreviglieri/pneumonia-xray-images>].
- [26] M. Elgendi, “The Effectiveness of Image Augmentation in Deep Learning Networks for Detecting COVID-19: A Geometric Transformation Perspective,” Accessed: September, 2021. [<https://www.frontiersin.org/articles/10.3389/fmed.2021.629134/full>].
- [27] S. Bouvaij, “Brain Tumor Classification (MRI),” Accessed: September, 2021. [<https://www.kaggle.com/sartajbhuvaji/brain-tumor-classification-mri>].
- [28] A. Aianzadeh, “Stanford CS231n- Dropout Assignment,” Accessed: September, 2021. [<https://medium.com/deepvision/cs231n-dropout-assignment-c80f3170854b>].
- [29] Ioffe, Sergey, Szegedy, and Christian, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” Accessed: September, 2021. [<https://arxiv.org/abs/1502.03167>].
- [30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks ,” Accessed: September, 2021. [<https://dl.acm.org/doi/pdf/10.1145/3065386>].

SEZIONE

- [31] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," Accessed: September, 2021. [<https://arxiv.org/pdf/1905.11946v1.pdf>].
- [32] P.-L. Prove, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," Accessed: September, 2021. [<https://towardsdatascience.com/mobilenetv2-inverted-residuals-and-linear-bottlenecks-8a4362f4ffd5>].
- [33] G. An, "The Effects of Adding Noise During Backpropagation Training on a Generalization Performance," Accessed: September, 2021. [<https://ieeexplore.ieee.org/document/6796981>].
- [34] Fawcett and Tom, "ROC Graphs: Notes and Practical Considerations for Researchers," 2004. [https://www.researchgate.net/publication/284043217_ROC_Graphs_Notes_and_Practical_Considerations_for_Researchers].