

Corso di Laurea Triennale in Ingegneria Informatica

Stima della pressione arteriosa attraverso tecniche di intelligenza artificiale

Martina Speciale

16 giugno 2023

relatori:

Ing. F. Pistolesi

Ing. M. Baldassini



Università di Pisa

Dipartimento di Ingegneria dell'Informazione

Anno Accademico 2022/2023

Indice

1	Introduzione	6
1.1	Obiettivo	6
1.2	Oggetto di Studio	6
1.2.1	Pressione Arteriosa	7
1.2.2	Attività elettrodermica	8
1.3	Intelligenza Artificiale	8
2	Le Reti Neurali	12
2.1	Il Percettrone	12
2.1.1	Funzioni di Attivazione	15
2.2	Costruire una Rete Neurale	18
2.2.1	Un modello semplificato del percettrone	18
2.2.2	Percettrone multi-output	19
2.2.3	Single Layer Neural Network	20
2.2.4	Deep Neural Network	22
2.3	Applicazione delle Reti Neurali	23
2.3.1	Quantificare la perdita	23
2.3.2	Binary Cross Entropy Loss	24
2.3.3	Mean Squared Error Loss	24
2.4	Addestramento delle Reti Neurali	25
2.4.1	Minimizzare la perdita	25
2.4.2	Gredient Descent	26
2.4.3	Back Propagation	27
2.5	Le Reti Neurali in Pratica : Ottimizzazione	28
2.5.1	Learning Rate Ideale	29
2.5.2	Adaptive Learning Rates	30
2.5.3	Un esempio pratico	30
2.5.4	Mini Batches	30
2.6	Overfitting	32
2.6.1	Regolarizzazione	33

3 Costruzione di un Modello	36
3.1 Reti Neurali: esempi pratici	36
3.2 Reti Neurali Convoluzionali	39
4 Raccolta Dati	43
4.1 Strumentazione	43
4.1.1 Misuratore di pressione - RS7 Intelli IT	43
4.1.2 Sensore GSR - Shimmer3 GSR+	43
4.2 Misurazioni	44
5 Modello per la stima della Pressione Arteriosa	46
5.1 Costruzione del database	46
5.2 Sviluppo della Rete Neurale	53
5.3 Pressione Arteriosa Sistolica	53
5.3.1 Preelaborazione dei dati	54
5.4 La Rete	56
5.4.1 Definire il modello	56
5.4.2 Metriche di Validazione del Modello	56
5.4.3 F1 Score	57
5.4.4 Matrice di Confusione	58
5.4.5 Modello Base	58
5.4.6 Modello Regolarizzato	62
5.5 Risultati	66
6 Future Applicazioni	71
6.1 Explainable AI	71
Bibliografia	74

Capitolo 1

Introduzione

1.1 Obiettivo

L’obiettivo del progetto illustrato nel seguito dell’elaborato è l’implementazione di un sistema non invasivo di misurazione per i valori relativi alla pressione arteriosa. A tal fine è stato sviluppato un modello basato sull’utilizzo di *reti neurali artificiali (ANNs : Artificial Neural Networks)*, una per la stima di ciascun valore di nostro interesse: pressione diastolica e sistolica), strumenti efficienti nell’investigare le relazioni presenti nei dati. Il sistema è pensato per produrre un risultato a partire da semplici immagini di parti del corpo specifiche (dorso della mano, polso e braccio - che si è pensato potessero offrire i risultati migliori).

1.2 Oggetto di Studio

L’ipertensione è una condizione che colpisce un miliardo di persone ed è il fattore di rischio di morte più comune al mondo.

Citando l’articolo *”L’impatto della pressione arteriosa sulla salute” ([02]), Istituto Superiore di Sanità :*

L’ipertensione arteriosa interessa, nel mondo, un adulto su tre ed è generalmente asintomatica sino al manifestarsi delle complicanze critiche. L’ipertensione è uno dei principali fattori di rischio delle malattie cardiovascolari e, secondo i dati forniti dall’Organizzazione mondiale della Sanità(Oms), è all’origine di oltre 9 milioni di decessi l’anno.

Blood Pressure Chart

Reference: Williams B. et al. 2018 ESC/ESH Guidelines for the management of arterial hypertension. European Heart Journal (2018) 39, 3021-3104

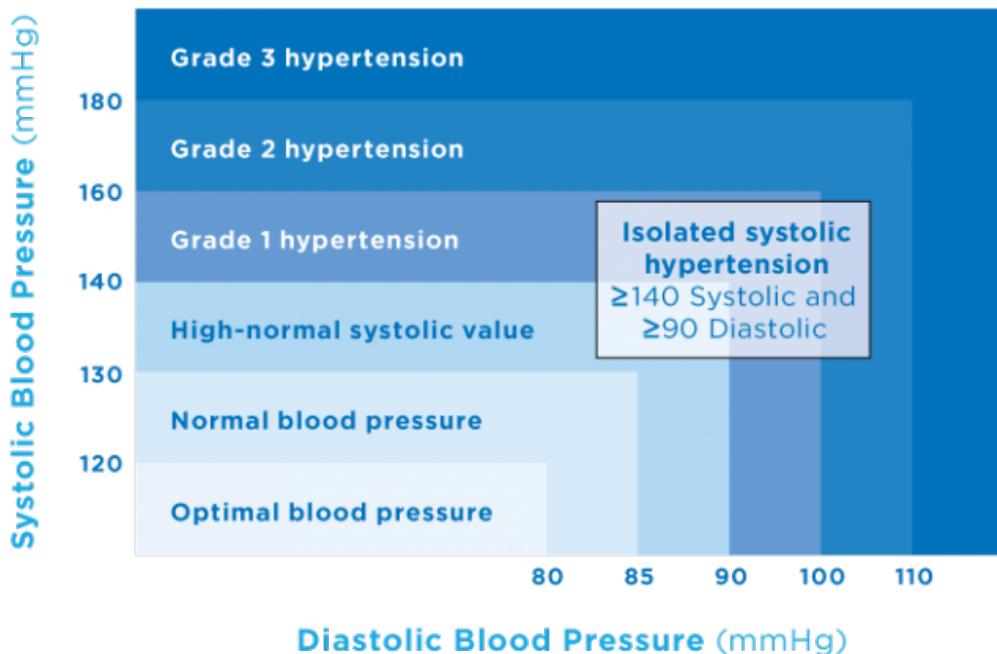


Figura 1.1: 2018 ESC/ESH ([00])

1.2.1 Pressione Arteriosa

La pressione arteriosa è la pressione che il cuore esercita per far circolare il sangue in tutto il corpo, si misura in millimetri di mercurio (*mmHg*) ed il suo valore è dato da due numeri, che rappresentano, rispettivamente:

- Pressione arteriosa sistolica : misurata al momento in cui il cuore si contrae e pompa il sangue nelle arterie
- Pressione arteriosa diastolica : misurata tra due contrazioni, mentre il cuore si rilassa e si riempie di sangue.

Il valore della pressione aumenta con lo sforzo, il freddo e quanto si è sottoposti a forti emozioni, diminuisce invece con il risposo e il sonno. Nella tabella 1.1, sono illustrati valori tipici relativi alla pressione, con relativa classificazione. Per le misurazioni della pressione nella fase di raccolta dati (Capitolo 4) si è fatto uso del misuratore di pressione da polso *RS7 Intelli IT* ([03]).

Pressione arteriosa [categoria]	Massima o sistolica (mmHg)	Minima o distolica (mmHg)
Normale	120 - 129	80 - 84
Pre-ipertensione	130 - 139	85 - 89
Ipertensione - stadio 1	140 - 159	90 - 99
Ipertensione - stadio 2	160 - 179	100 - 109
Ipertensione - stadio 3	≥ 180	≥ 110
Ipertensione sistolica isolata	≥ 140	< 90

Tabella 1.1: valori tipici relativi alla pressione arteriosa

1.2.2 Attività elettrodermica

L'attività elettrodermica, nota anche come *risposta galvanica cutanea (galvanic skin response, GSR)*, indica la misura delle continue variazioni nelle proprietà di conduttanza elettrica della pelle, in risposta alla secrezione di sudore. La *risposta galvanica cutanea*, così come l'attività sudomotoria sono modulate autonomamente dall'attività del *sistema nervoso simpatico*, una componente del sistema nervoso autonomo che interviene nel controllo delle funzioni corporee involontarie. La *conduttanza elettrica* (reciproco della *resistenza*), indica la misura in cui un passaggio di corrente è favorito. Il segnale GSR presenta due componenti

- SCL (*skin conductance level*) : considera il livello di conduttanza cutanea. Si tratta di una componente tonica, che varia lentamente.
- SCR (*skin conductance response*) : considera la risposta di conduttanza cutanea. È la componente fasica, che varia più rapidamente ed è correlata a stimoli esterni e di natura non specifica.

Per le rilevazione dei valori GSR si è fatto uso del sensore *Shimmer3 GSR+* ([04])

1.3 Intelligenza Artificiale

Intelligenza Artificiale, Machine Learning e Deep Learning (Figura 1.2) rappresentano le radici del campo delle *Data Science*, in continua crescita. Nell'era dei *Big Data*, l'Intelligenza Artificiale si pone il fine ultimo di impartire alle macchine l'intelligenza che generalmente attribuiamo al genere umano facendo uso di grandi quantità di dati. Nella breve introduzione al campo dell'AI facciamo riferimento, nel seguito, a qualche passaggio del libro *La Scocciatoia* (N. Cristianini, [01]). Quando si parla di *machine learning* (apprendimento automatico) si entra in un campo volto allo studio ed alla realizzazione di macchine che migliorano il proprio comportamento con l'esperienza. Con la parola "apprendimento" ci riferiamo a quelle situazioni in cui la macchina acquisisce nuove abilità,

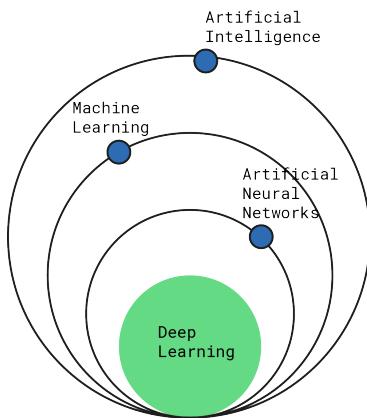


Figura 1.2: Paradigma di AI, machine learning e deep learning

o migliora quelle già esistenti, senza essere stata esplicitamente programmata. Tutti gli agenti intelligenti in uso oggi si basano su qualche forma di apprendimento automatico: è questo che consente loro di affrontare non solo ambienti mutevoli e incerti, ma anche compiti per cui non esiste una chiara descrizione teorica.

"L'irragionevole efficacia dei dati"

Nel 2009 un gruppo di ricercatori di Google pubblicò un articolo che sarebbe diventato una sorta di manifesto nel campo dell’Intelligenza Artificiale. Era intitolato "L’irragionevole efficacia dei dati" (*The Unreasonable Effectiveness of Data*, [05]), un gioco di parole su un classico articolo scritto nel 1960 da Eugene Wigner sul sorprendente potere della matematica. L’articolo celebrava il potere dei dati - definiti come "il miglior alleato di cui disponiamo" - nel plasmare il comportamento intelligente e codificava quelle che ormai erano divenute delle pratiche comuni nel campo dell’Intelligenza Artificiale. Quegli anni furono infatti fondamentali per finalmente abbracciare l’approccio *data-driven*, che poggia le sue fondamenta sull’idea di base che *devono essere i dati, e non i modelli o le regole, a guidare il comportamento degli agenti intelligenti*.

Un comportamento proteso al raggiungimento di obiettivi ha senso solo in un ambiente che è, almeno in parte, controllabile ed osservabile, ovvero "in cui azioni appropriate aumentano le possibilità di successo". Un assunto ancora più forte alla base di ogni apprendimento e generalizzazione è che azioni simili risultino in conseguenze simili: è questo tipo di regolarità che permette ad un agente di affrontare una varietà infinita di situazioni con conoscenze finite, incluse situazioni mai viste e talvolta anche ambienti totalmente nuovi. Anche deboli regolarità statistiche possono essere sufficienti per conferire qualche vantaggio all’agente, e imparare tali regolarità dall’esperienza è uno dei modi in cui si produce un comportamento intelligente. La regola è quella di *scoprire e sfruttare l’ordine dell’ambiente su cui si lavora*.

La programmazione tradizionale parte da una serie di regole ben precise, definite da chi programma, le quali agiranno su dei dati passati in ingresso in maniera tale da poter ottenere un qualche risultato o una qualche risposta. Queste generiche regole, espresse in un linguaggio di programmazione, rappresentano la maggior parte di qualunque codice si scriva quando si programma tradizionalmente. Se si capovolge questo paradigma, fornendo in ingresso risposte e dati e lasciando alla macchina l'onere di estrapolare quelle regole prima definite dal programmatore, si entra nel campo del *machine learning*. È questo il cuore di ciascun processo basato sull'apprendimento autonomo: a partire da un insieme di dati che presentano una serie di pattern comuni e intrinseci, si fa in modo che il modello impari a riconoscerli.

Machine Learning e *ottimizzazione matematica* sono campi profondamente legati. Molti problemi di apprendimento automatico sono formulati in termini di minimizzazione di una funzione di perdita, con riferimento ad un determinato *training set*. Tale funzione, come vedremo in dettaglio nel corso del Capitolo 2, rappresenta la differenza tra i valori predetti dal modello in fase di addestramento e i valori attesi per ciascuna istanza in esempio. L'obiettivo finale è di impartire al modello -grazie ad un processo di generalizzazione- la capacità di predire correttamente i valori attesi su un set di istanze mai visto prima, il cosiddetto *testing set*, grazie alla minimizzazione della funzione di perdita su questo insieme di istanze. A seconda del *feedback* su cui si basa il sistema di *machine learning*, distinguiamo tre differenti categorie:

- *Apprendimento supervisionato* : vengono presentati al modello degli input di esempio, con relativi output desiderati, al fine di apprendere una regola generale in grado di mappare correttamente gli input nei rispettivi output. Questo scenario è quello di interesse per il lavoro svolto nell'elaborato.
- *Apprendimento non supervisionato* : alla macchina vengono forniti dati in input, senza alcun output atteso, allo scopo di apprendere una qualche struttura nei dati presentati in ingresso.
- *Apprendimento con rinforzo* : la macchina interagisce con un ambiente dinamico nel quale deve raggiungere un certo *goal*. Esplorando il dominio del problema, alla macchina vengono forniti dei feedback in termini di ricompense o penitenze, in modo da indirizzarla verso la soluzione migliore.

Se anziché porre la propria attenzione sul *feedback*, ci si concentra invece sul tipo di *output* atteso, si parla invece di:

- *Classificazione*, durante la quale gli input sono divisi in classi e il sistema di apprendimento deve produrre un modello in grado di assegnare ad un input una o più classi tra quelle disponibili. Questi tipi di task sono affrontati mediante tecniche di apprendimento supervisionato. Si parla di classificazione quando a ciascuna immagine data in pasto al modello, viene assegnata una particolare etichetta (nel seguito vedremo ad esempio in prima istanza la suddivisione dei valori relativi alla pressione in tre classi: bassa, normale e alta)
- *Regressione*, concettualmente simile alla classificazione, con la differenza che l'output ha un dominio continuo e non discreto. Anche qui si va di pari passo con l'apprendimento supervisionato. Rientriamo in questo campo quando il dominio dell'output è virtualmente infinito e non limitato ad un certo insieme discreto di possibilità.
- *Clustering*, nel quale, come nella classificazione, un insieme di dati viene diviso in segmenti che tuttavia non sono noti a priori. La natura stessa dei problemi appartenenti a questa categoria li rende tipicamente dei task di apprendimento non supervisionato.

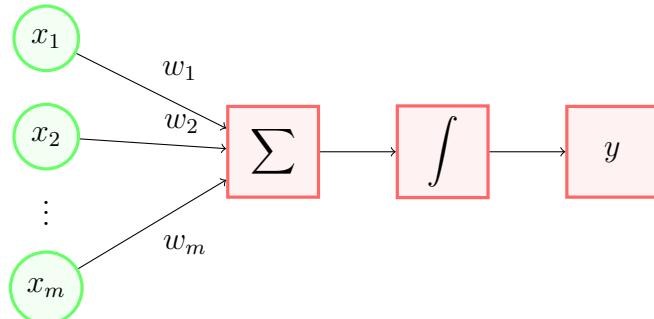
Capitolo 2

Le Reti Neurali

L'intero capitolo segue liberamente la prima lezione del corso [06], *"Introduction to Deep Learning"*.

2.1 Il Percettrone

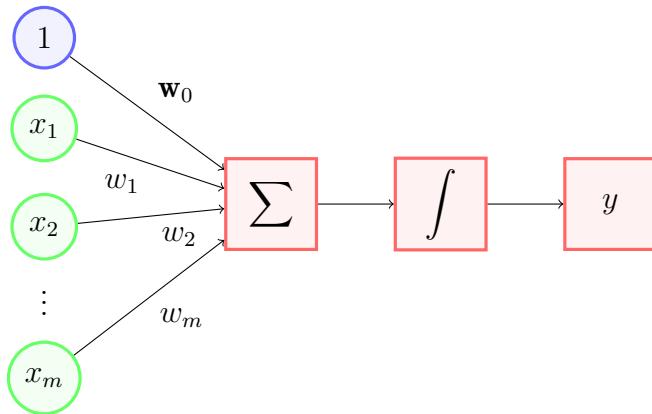
L'elemento costitutivo strutturale di ogni rete neurale è un singolo neurone. Nel linguaggio delle reti neurali si parla di *percettrone*. Con riferimento alla figura sottostante, raffigurante la struttura concettuale del percettrone, consideriamo una serie di m input $\{x_i\}_{i=1}^m$.



$$\begin{aligned} z' &= \sum_{i=1}^m x_i \cdot w_i \\ y &= g \left(\sum_{i=1}^m x_i \cdot w_i \right) \end{aligned}$$

Ogni input x_i viene moltiplicato per il corrispettivo peso w_i . Sommando tutti questi prodotti si ottiene z' , che viene passato ad una funzione non lineare g , detta *funzione di attivazione* (*non-linear activation function*). Ciò produce l'output finale y del percettrone.

Sommendo z' ad uno scalare che prende il nome di *bias* - che indichiamo con w_0 - si ottiene un risultato z . Il *bias* può essere pensato come un peso regolabile corrispondente ad un input unitario e viene utilizzato allo scopo di aumentare l'accuratezza nella classificazione del modello, addizionandolo alla somma pesata z' . Il risultato z viene poi, come prima, passato a g , *funzione di attivazione* (non lineare), che produce y .



$$y = g \left(w_0 + \sum_{i=1}^m x_i \cdot w_i \right)$$

Il quadro appena tracciato permette alla rete, sottoposta a diversi input, di muoversi in maniera non lineare grazie all'applicazione della funzione di attivazione.

Al fine di semplificare la notazione definiamo il vettore X , contenente i valori di tutti gli m input ed il vettore W , contenente i valori di tutti gli m pesi.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix}, \quad W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

Si considera il prodotto scalare tra i due, cui viene sommato il *bias* w_0 , ottenendo così z . Applicando il risultato z all'applicazione non lineare g si ottiene l'output y del percettrone.

$$y = g(w_0 + X^T \cdot W)$$

Vediamo di seguito un semplice esempio numerico, per meglio comprendere la struttura del percettrone.

Consideriamo $w_0 = 1$ e $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

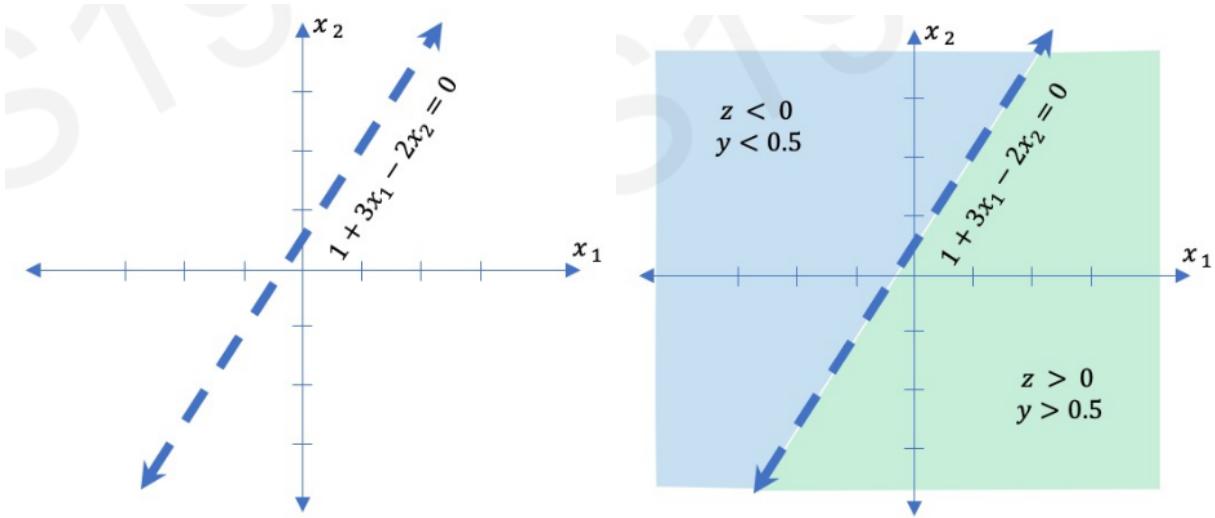


Figura 2.1: retta separatrice, di confine (da [06])

$$\begin{aligned} y &= g(w_0 + X^T W) \\ &= g \left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix} \right), \\ y &= g(1 + 3x_1 - 2x_2) \end{aligned}$$

z , la componente all'interno della nostra funzione g non lineare è, nel caso in esame, di fatto una retta su un piano. La rete neurale presa in esempio, costituita da un singolo percettrone, ha infatti solo due input e se consideriamo lo spazio di tutti i possibili input che la rete può vedere, si può tracciare un confine di decisione, disegnando la retta che suddivide il piano in due semipiani, individuata da

$$z = 1 + 3x_1 - 2x_2$$

Se come input, ad esempio, consideriamo il vettore

$$X = \begin{bmatrix} -1 \\ 2 \end{bmatrix},$$

è facile vedere come questo *data point* viva nel semipiano che risulta in uno z negativo, che a sua volta verrà mappato, attraverso g (prendendo ad esempio come *funzione di attivazione* la *funzione sigmoidea*, che approfondiamo più avanti), in un particolare output $y < 0.5$. La retta di confine individuata da z vive nel mezzo, dunque a seconda della posizione dell'input rispetto ad essa, l'applicazione $g(z)$ produrrà un tipo di risultato piuttosto che un altro. Ora, sopra abbiamo preso in esame un semplice esempio, utile per visualizzare chiaramente quello che è un concetto base della struttura analizzata.

Ci troviamo infatti a lavorare su un piano ed è pertanto semplice e possibile tracciare e visualizzare la retta di confine. Ovviamente, per quasi la totalità dei problemi di nostro interesse (a partire dal particolare caso preso in esame in questo scritto), i *data points* non saranno mai bidimensionali. Basti pensare ad un'applicazione che prende come input delle immagini: la dimensione di un'immagine corrisponde al numero di pixel che la compongono, dunque si parla di input che vivono in spazi la cui dimensione può essere di ordini arbitrariamente grandi, dalle migliaia sino ai milioni ed oltre. Disegnare grafici del genere non è chiaramente possibile. L'esempio sopra, sebbene non realistico o particolarmente rappresentativo, ci permette di ridurre il problema ed utilizzare l'intuizione geometrica per estrapolare comportamenti che poi si ritrovano su scala molto più larga, applicati a modelli molto più complessi e aderenti alla realtà.

2.1.1 Funzioni di Attivazione

Vi sono diversi tipi di funzioni di attivazione non lineare, popolari nel mondo delle reti neurali. Tra le più comuni, ritroviamo:

- *Sigmoid Function* (funzione sigmoidea, in figura 2.2)

```
1     tf.math.sigmoid(z)
2
```

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

Le funzioni sigmoide sono comunemente utilizzate come funzioni di attivazione nell'ambito delle reti neurali per introdurre una componente di non linearità nel modello e per assicurarsi di muoversi all'interno di specifici intervalli. La semplice relazione polinomiale tra la derivata e la funzione stessa risulta oltretutto semplice da implementare, dal punto di vista informatico. Può essere pensata come una versione continua della funzione gradino: assume valori compresi tra 0 ed 1, prendendo come input qualsiasi valore sull'asse dei numeri reali. Assume particolare importanza quando si fa utilizzo, per esempio, di distribuzioni di probabilità, dato il codominio delle probabilità stesse, che sappiamo vivere tra 0 ed 1.

- *Hyperbolic Tangent* (tangente iperbolica, in figura 2.3)

```
1     tf.math.tanh(z)
2
```

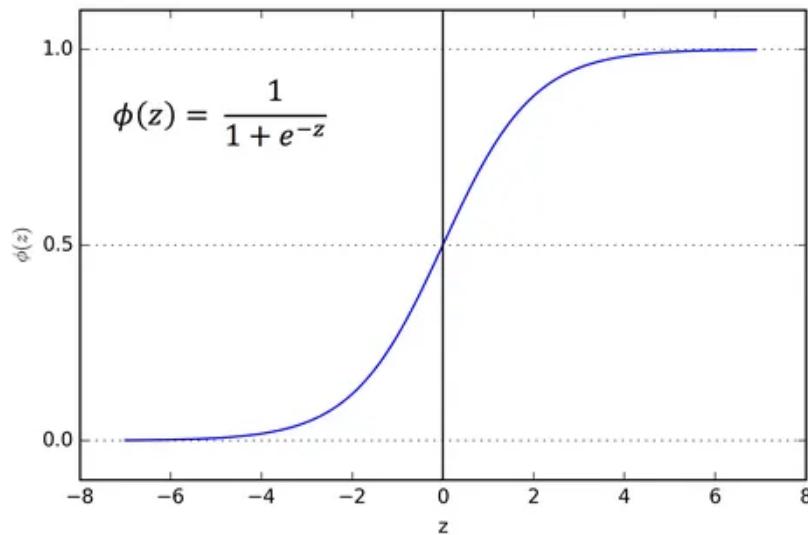


Figura 2.2: Sigmoid Function

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

La *tangente iperbolica*, molto simile alla funzione sigmoide, è centrata nello zero ed ha un intervallo compreso tra -1 e 1 . È continua e differenziabile ovunque.

- *Rectified Linear Unit* (ReLU, rettificatore, in figura 2.4)

```
1   tf.nn.relu(z)
2
```

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{altrimenti} \end{cases}$$

Particolarmente efficiente nel contesto delle moderne *reti neurali profonde* (*Deep Neural Networks*), di cui è stata riconosciuta come la funzione di attivazione più largamente utilizzata, il *rettificatore (ReLU)* è una funzione definita come la parte positiva del suo argomento. La funzione rettificatore, in confronto alla più diffusa tangente iperbolica, permette un addestramento della rete più efficiente in più breve tempo, facendo utilizzo di grandi e complessi insiemi di dati. Lineare a tratti, la ReLU è infatti estremamente efficiente da calcolare, specialmente quando il calcolo è di tipo derivativo (derivate costanti, ad eccezione che in 0).

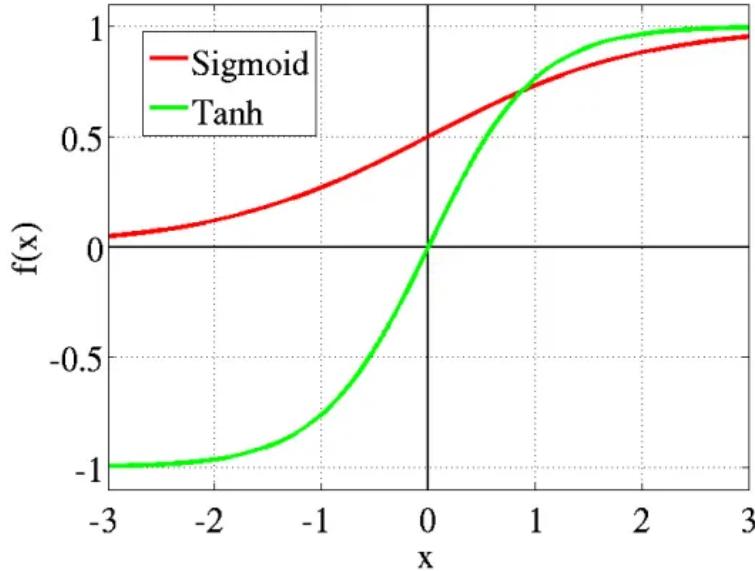


Figura 2.3: Funzione sigmoide e tangente iperbolica a confronto

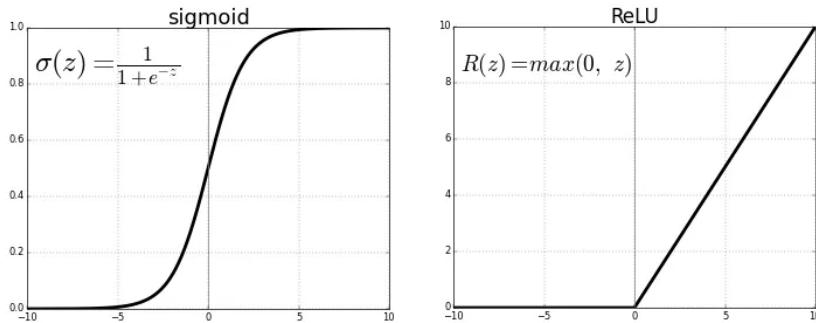


Figura 2.4: Funzione sigmoide e rettificatore a confronto

Ci si può chiedere perché ci si debba effettivamente servire di una funzione di attivazione non lineare, che sostanzialmente va a complicare l'immagine che viene posta del problema. Il punto chiave che spiega e rende indispensabile l'utilizzo di funzioni non lineari risiede nella necessità di dover associare una qualità di *non linearità* ai nostri dati. Se pensiamo ai dati infatti, quasi tutti quelli per cui nutriamo interesse - tutti i dati provenienti dal mondo reale fondamentalmente - sono altamente non lineari. Volendo essere in grado di lavorare su *data set* del genere, affetti da alta non linearità, risulta dunque chiara la necessità di disporre di modelli che rispecchino quella stessa non linearità, in maniera tale da poter riconoscere ed estrarre dei pattern nei dati analizzati. Immaginiamo di avere il *data set* raffigurato dal grafico in figura 2.5 e di voler separare i punti rossi da quelli verdi. Distinguere i due tipi di dati può sembrare un compito estremamente semplice, ma se ci ponessimo il limite di poter usare a tal fine una singola linea retta, ecco che il

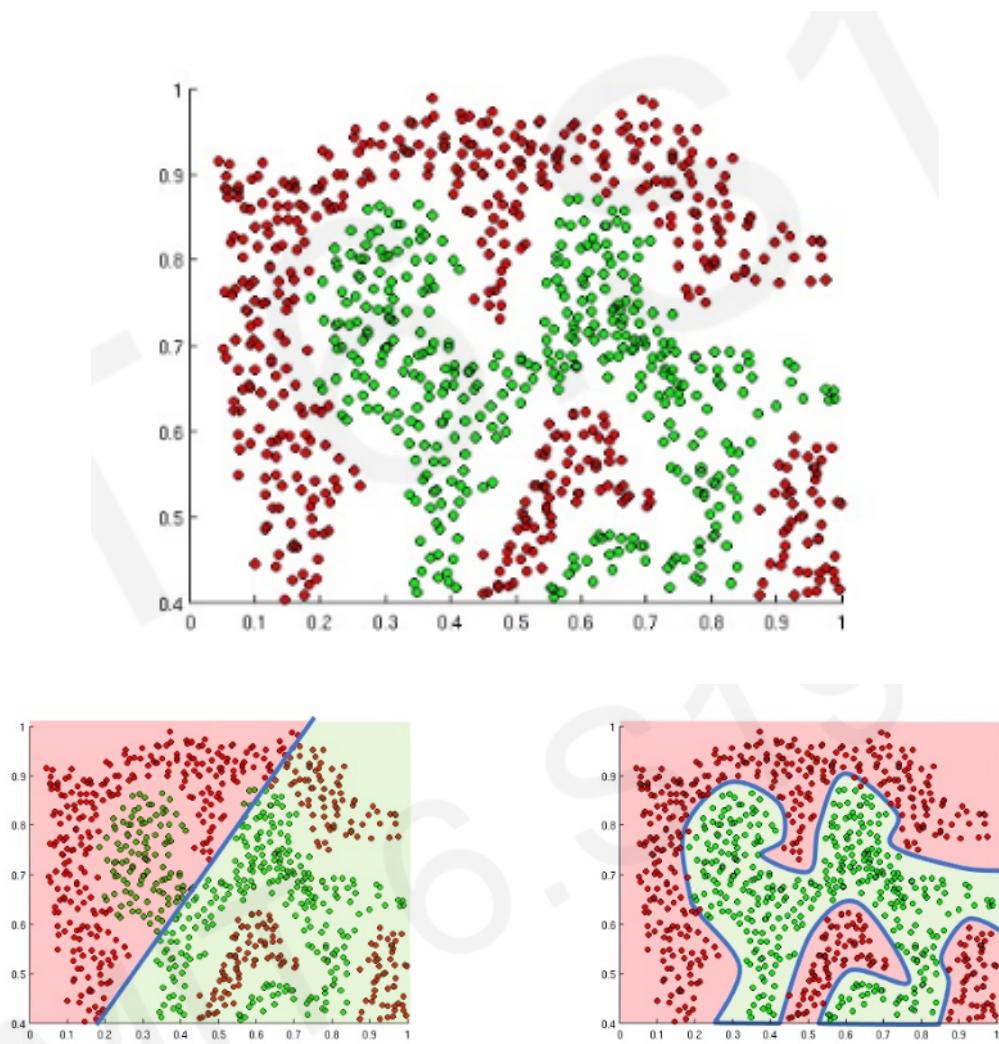


Figura 2.5: Esempio (da [06])

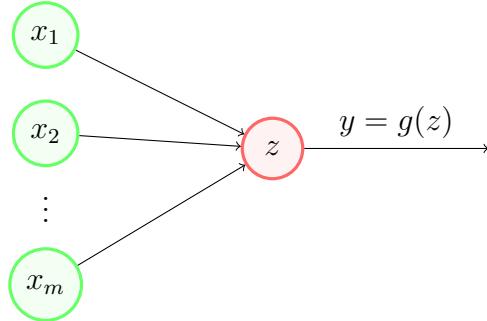
problema risulta non più risolvibile in maniera efficiente. L'introduzione di applicazioni non lineari permette di superare un ostacolo del genere. Le funzioni di attivazione dunque sono necessarie dal momento che permettono di lavorare *con e su* tipi di dati non lineari. È proprio questo fattore di non linearità a rendere le reti neurali così potenti alla base.

2.2 Costruire una Rete Neurale

2.2.1 Un modello semplificato del percepitrone

Dall'idea di partenza del percepitrone appena illustrata, tentiamo ora di chiarire come da questo singolo neurone si possa iniziare a costruire qualcosa di più complicato: una completa rete neurale sulla quale realizzare il nostro modello. Da qui in avanti faremo uso di un diagramma leggermente semplificato del percepitrone. Assumiamo, nella figura

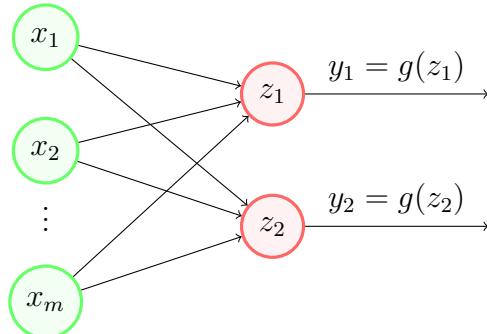
sotto, che i pesi w_0, \dots, w_m , non più raffigurati, siano associati a ciascuna linea. Anche il *bias*, w_0 , è stato rimosso per semplicità, ma assumiamo che sia sempre presente e legato ad un simbolico input unitario.



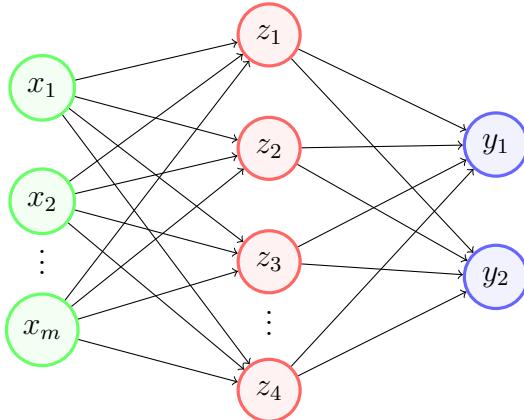
Continuiamo a chiamare z il risultato del prodotto scalare sommato al *bias*. Questo risultato può essere pensato come al rappresentante primo dello stato di un particolare neurone in un dato momento, da passare alla funzione di attivazione g . L'output è $g(z)$.

2.2.2 Percettrone multi-output

Il nostro prossimo obiettivo è quello di definire l'output di una rete neurale multi strato (*multi-layered neural network*). A tal fine iniziamo pensando di volere, ad esempio, due output anzichè uno: bisogna semplicemente attrezzare la nostra rete di due percetroni. Ogni percettrone controllerà un output, avendo in ingresso i medesimi input. Va da sé che il numero di output ottenibili da un singolo strato di quella che sarà la nostra rete neurale può essere arbitrario.



2.2.3 Single Layer Neural Network



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^m x_j \cdot w_{j,i}^{(1)}$$

$$y_i = g \left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} g(z_j) \cdot w_{j,i}^{(2)} \right)$$

Poichè tutti gli input sono densamente connessi a tutti gli output, gli strati che si vengono a creare all'interno della rete neurale sono chiamati *dense layers* (strati densi). I *dense layers* creano interconnessioni tra i vari strati della rete. Ogni neurone è connesso ad ogni altro neurone dello strato immediatamente successivo, il che significa che il suo valore di output $g_i(z)$ diventa input per i neuroni che lo seguono. Grazie al modello matematico descritto, rappresentante della funzione del singolo percettrone, è possibile iniziare a costruire la nostra prima rete neurale a partire da zero.

Dense Layer

```

1  class MyDenseLayer(tf.keras.layers.Layer):
2      def __init__(self, input_dim, output_dim):
3          super(MyDenseLayer, self).__init__()
4
5          #Initialize weights and bias
6          self.W = self.add_weight([input_dim, output_dim])
7          self.b = self.add_weight([1, output_dim])
8
9      def call(self, inputs):
10         #Forward propagate the inputs
11         z = tf.matmul(inputs, self.W) + self.b
12
13         #Feed through a non-linear activation
14         output = tf.math.sigmoid(z)
15

```

```

16     return output
17

```

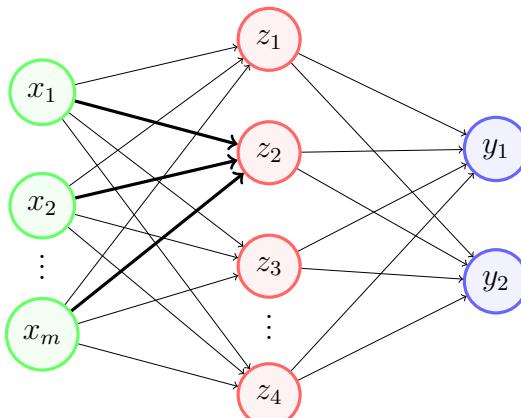
Per prima cosa inizializziamo i due vettori relativi a pesi e bias (linee 6-7 del codice): sono questi i fattori che verranno modificati durante l'addestramento, non potendo chiaramente avere un reale controllo sugli input. Dopo aver definito questi parametri dello strato, rimane da definire come funziona la propagazione di informazioni da uno strato a quello successivo (*forward propagation*). Definiamo a tal fine la funzione *call* (linea 9). Il concetto è esattamente lo stesso discusso sino ad ora: prodotto scalare di input e pesi, cui viene sommato il *bias* (linea 11). Alla linea 14 vediamo poi l'applicazione della *funzione di attivazione non lineare*, che produce il risultato finale. Tale codice definisce un intero strato della rete neurale ed è disponibile in diverse librerie, come *tensorflow*. Riprendendo lo schema illustrato sopra del *percettrone multi-output*, per definire ad esempio il modello a soli due output (come illustrato in figura), basterà eseguire :

```

1 import tensorflow as tf
2 layer = tf.keras.layers.Dense(units = 2)

```

Compresa l'esecuzione che porta alla realizzazione di un singolo strato si può iniziare ora a pensare a come poter *impilare* strati differenti. Disponiamo di una trasformazione che ci porta essenzialmente da una serie di input ad un output nascosto, bisogna dunque pensare a come poter definire un modello da seguire per trasformare quegli input in qualche nuovo spazio dimensionale, al fine di raggiungere idealmente un risultato più vicino al valore che vogliamo predire. Questa trasformazione, o meglio la comprensione del modello che trasforma diversi input in output desiderati, verrà infine appresa (ed affrontata, più avanti). Per il momento il punto sul quale focalizzarci è comprendere come, anche disponendo ora di queste reti neurali molto più complesse rispetto al singolo percettrone analizzato, queste in realtà non rappresentino nulla di concettualmente più complesso rispetto a ciò che si è già visto.



Se ci focalizziamo su un unico neurone nella rete, ad esempio z_2 , la sua funzione sarà totalmente compatibile con ciò che abbiamo descritto sino ad ora.

$$\begin{aligned} z_2 &= w_{0,2}^{(1)} + \sum_{j=1}^m x_j \cdot w_{j,2}^{(1)} \\ &= w_{0,2}^{(1)} + x_1 \cdot w_{1,2}^{(1)} + x_2 \cdot w_{2,2}^{(1)} + \dots + x_m \cdot w_{m,2}^{(1)} \end{aligned}$$

Il suo output è ottenuto eseguendo il prodotto scalare e sommandolo al bias prima di passarlo alla funzione di attivazione. Se guardiamo ad un nodo diverso nella rete, per esempio z_3 , il discorso sarà esattamente lo stesso. A cambiare, a fronte dei medesimi input, saranno i pesi applicati a quegli input. Otterremo da z_3 un output differente rispetto a z_2 , ma il principio matematico alla base del singolo percettrone è esattamente lo stesso.

2.2.4 Deep Neural Network

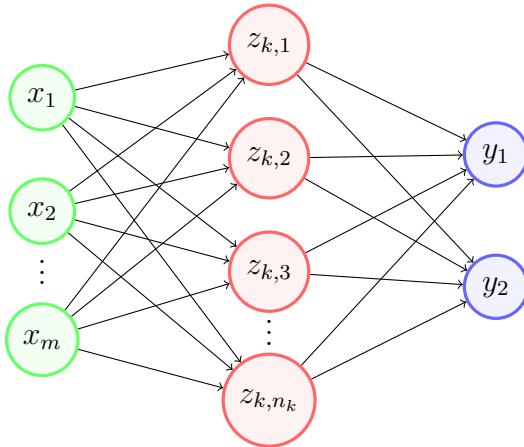
Volendo sovrapporre questo tipo di soluzioni (*layers*), attraverso il codice si può andare ben oltre la definizione di un singolo strato: con estrema semplicità si può arrivare a realizzare quelli che vengono chiamati *sequential models* (modelli sequenziali).

```

1 import tensorflow as tf
2
3 model = tf.keras.Sequential([
4     tf.keras.layers.Dense(n),
5     tf.keras.layers.Dense(m),
6     # ...
7     tf.keras.layers.Dense(2)
8 ])

```

Si definisce un layer dopo l'altro e ciascuno di questi strati va a definire, fondamentalmente, la *forward propagation* di informazioni, ora non solo a livello del neurone, ma a livello dello strato. Ogni layer è completamente connesso a quello successivo: gli output di ciascun layer non sono altro che gli input del successivo. Volendo creare una cosiddetta *deep neural network*, basta continuare a impilare questi strati, andando a costruire una sorta di modello gerarchico. Reti neurali del genere sono semplicemente reti cui l'output finale viene calcolato partendo dall'input per andare sempre più in profondità, passando di layer in layer, in questa progressione di strati. L'ultimo rappresenta il cosiddetto *output layer* e porta con sé la predizione finale.



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) \cdot w_{j,i}^{(k)}$$

2.3 Applicazione delle Reti Neurali

Abbiamo visto come passare da un singolo neurone ad un layer, fino ad arrivare alla realizzazione di una *deep neural network*. Vediamo ora come possono effettivamente essere utilizzate le reti neurali appena descritte, ed in particolare come poter usare i principi discussi per poter risolvere problemi reali. Una rete neurale "appena nata" non ha alcuna informazione sul mondo o sull'ambiente che deve analizzare e per la quale è stata realizzata. Si può pensare alla rete, in questo stato primordiale, come ad un bambino: non ha imparato ancora nulla, dunque il nostro primo compito è quello di addestrarla. Una parte fondamentale di questo processo di apprendimento sta nel definire e comunicare alla rete quando compie degli errori.

2.3.1 Quantificare la perdita

La perdita (*loss*) di una rete misura il costo sostenuto dalle nostre predizioni errate. Matematicamente dovremmo pensare a come poter rispondere ad una simile domanda: "la rete neurale ha compiuto un errore? E se è questo il caso, come posso comunicarle quanto è effettivamente grande questo errore in modo tale che la prossima volta che vede dati e informazioni di questo tipo (il medesimo *data point*) possa migliorarsi, minimizzando quell'errore appena compiuto e osservato?". Nel linguaggio delle reti neurali questi errori prendono il nome di *perdite*. Ciò che vogliamo fare è definire la cosiddetta *funzione di perdita* (*loss function*, anche detta *objective function*, *cost function* o *empirical risk*), che prenderà come input la nostra predizione ed il valore effettivamente prodotto dalla rete,

al fine di addestrare la rete neurale, impartendole una conoscenza circa la misurazione dell'errore che compie, così che possa migliorarsi.

$$\mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

dove $f(x^{(i)}; W)$ rappresenta il valore da noi predetto e $y^{(i)}$ è invece il risultato prodotto dalla rete neurale. La distanza tra questi due valori fornisce un'importante informazione sulla perdita osservata. L' *empirical loss* $J(W)$ misura la perdita totale sull'intero *data set*.

$$J(W) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

L'obiettivo finale è, considerando l'intero *data set* anziché un singolo *data point*, minimizzare in media tutti gli errori che la rete neurale compie.

2.3.2 Binary Cross Entropy Loss

Se guardiamo ad un problema binario, il cui output può assumere solo due valori (0 o 1, si parla di *binary classification*), possiamo usare una funzione di perdita nota come *loss function of the Softmax Cross Entropy Loss*. La nozione di entropia incrociata (*cross entropy*), è stata sviluppata al MIT da Claude Shannon e rappresenta una nozione fondamentale nell'abilità sviluppata sino ad oggi nell'addestrare le reti neurali.

$$J(W) = -\frac{1}{n} \sum_{i=1}^n y^{(i)} \log(f(x^{(i)}; W)) + (1 - y^{(i)}) \log(1 - f(x^{(i)}; W))$$

```
1 loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y,
                                                               predicted))
```

2.3.3 Mean Squared Error Loss

Se invece che predire il risultato di un problema binario (il cui output è dunque binario), si vuole predire un risultato che prevede un output rappresentato da una variabile continua, si può fare uso dell'errore quadratico medio (*mean squared error loss*), che indica la discrepanza quadratica media fra i valori dei dati osservati ed i valori dei dati stimati.

$$J(W) = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - f(x^{(i)}; W))^2$$

```
1 loss = tf.reduce_mean(tf.square(tf.subtract(y, predicted)))
2 loss = tf.keras.losses.MSE(y, predicted)
```

2.4 Addestramento delle Reti Neurali

Siamo ora pronti per mettere insieme tutte le informazioni viste sino ad ora ed affrontare il problema dell’addestramento di una rete neurale. Non ci limiteremo ad individuare l’errore compiuto dalla rete, stimandone la perdita, ma, cosa più importante, andremo a minimizzare quella perdita grazie all’utilizzo di grandi quantità di *training data*.

2.4.1 Minimizzare la perdita

Vogliamo una rete neurale che minimizzi la *perdita empirica* (o *empirical risk*), mediamente su tutto il nostro *data set*. Matematicamente ciò si traduce nel cercare la matrice W che minimizzi $J(W)$, dove $J(W)$ rappresenta la *loss function* sull’intero *data set* e W l’insieme dei pesi nella rete. Il nostro obiettivo è dunque quello di trovare il set di pesi che in media andrà a risultare nella più piccola perdita possibile.

$$\begin{aligned} W^* &= \arg \min_W \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; W), y^{(i)}) \\ W^* &= \arg \min_W J(W) \end{aligned}$$

con

$$W = \{W^{(0)}, W^{(1)}, \dots\}$$

Potremmo avere centinaia di pesi e dunque una rete neurale molto piccola o, nelle reti neurali di oggi, si potrebbero avere miliardi o trilioni di pesi. L’obiettivo è sempre quello di trovare il valore di ciascun singolo peso in maniera tale che la perdita sia minima. $J(W)$, la nostra funzione di perdita, è semplicemente una funzione calcolata su W e dunque, per ogni istanza dei pesi, possiamo calcolare uno scalare, $J(W)$ che rispecchi ”di quanto sbaglierebbe” la nostra rete neurale per quella particolare istanziazione dei pesi. Proviamo ancora una volta a visualizzare la situazione attraverso un semplice esempio, bidimensionale, prendendo in esame una rete neurale molto piccola, che disponga di soli due pesi. Il nostro obiettivo è quello di trovare i valori dei pesi ottimali al fine di addestrare questa rete neurale. $J(W)$ valuta in ogni punto dello spazio il valore di perdita per ogni singola possibile istanziazione dei due pesi w_0 e w_1 . Idealmente ciò che vorremmo è trovare il minimo globale nel grafico in figura 2.6, o meglio i valori di W , qui solo due, che realizzano quel minimo. Come illustrato in figura, viene scelto in maniera aleatoria un punto iniziale $P = (w_0, w_1)$. Da qui viene valutata la rete neurale, calcolando la perdita relativa a questa specifica configurazione di pesi. È poi necessario andare a valutare come la perdita vari in un intorno di P , calcolando il gradiente della funzione, $\nabla J(W)$. Il gradiente indica la direzione della retta di massima pendenza tangente al grafico nel punto P ed il suo verso punta nella direzione di un punto di massimo locale. Volendo noi

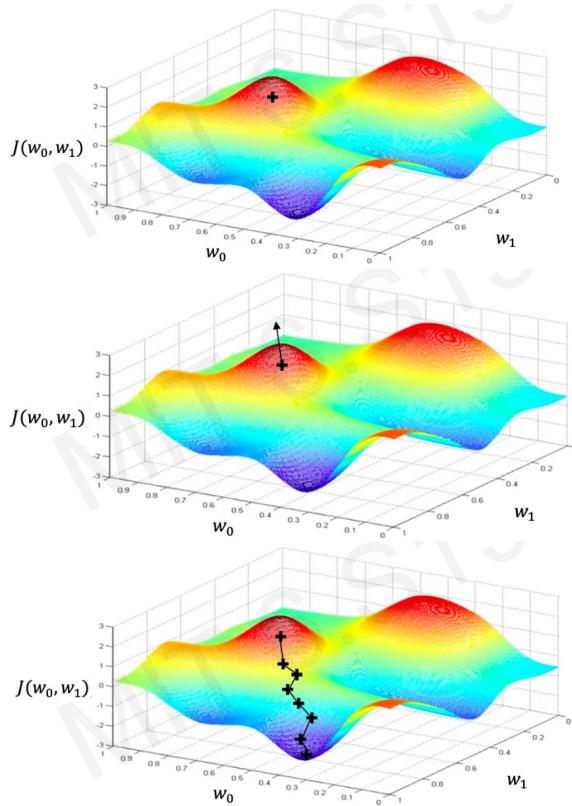


Figura 2.6: funzione di perdita J calcolata in funzione di w_0 e w_1 (da [06])

minimizzare la perdita ci sposteremo in direzione opposta rispetto al gradiente tramite l'azione del vettore $-\eta \frac{\partial J(W)}{\partial W}$, dove η è uno scalare la cui entità sarà approfondita più avanti. Iterando il processo, andando a valutare la rete neurale tramite la valutazione di $J(W)$ nei punti ottenuti di volta in volta, continuiamo a spostarci sul grafico convergendo, idealmente, al minimo.

2.4.2 Gradient Descent

Possiamo così riassumere l'algoritmo appena discusso, conosciuto formalmente come *discesa del gradiente (gradient descent)*.

1. Inizializzare i pesi della rete, assegnandogli valori casuali $\sim \mathcal{N}(0, \sigma^2)$
2. Fino a che non converge, loop:
 - Calcolo del gradiente $\nabla J(W) = \frac{\partial J(W)}{\partial W}$
 - Aggiornamento dei pesi $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
3. Ritorno dei pesi

η è noto come tasso di apprendimento (*learning rate*). Possiamo pensare al tasso di apprendimento come un piccolo valore che esprime la fiducia posta nel gradiente, così da definire il passo di aggiornamento ottimale nella direzione ad esso opposta. L'algoritmo sopra descritto è facilmente implementabile, come segue :

```

1 import tensorflow as tf
2
3 weights = tf.Variable([tf.random.normal()])
4
5 while True: #loop forever
6     with tf.GradientTape() as g:
7         loss = compute_loss(weights)
8         gradient = g.gradient(loss, weights)
9
10    weights = weights - lr * gradient

```

Andiamo ora brevemente ad affrontare più nello specifico il discorso relativo all'utilizzo e alla funzione del gradiente nel metodo appena esposto, punto critico nell'addestramento delle reti neurali. È grazie al gradiente infatti se le reti neurali sono in grado di apprendere se hanno bisogno di muovere i propri pesi ed in che direzione. Se non fossimo in grado di calcolare il gradiente allora addestrare una rete neurale non sarebbe possibile.

```

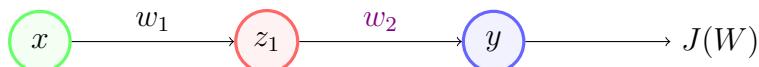
1 gradient = g.gradient(loss, weight)

```

il calcolo di questa linea di codice (sopra alla linea 8) è noto come *back propagation*

2.4.3 Back Propagation

La retropropagazione dell'errore (*backpropagation*) è un algoritmo per l'addestramento delle reti neurali artificiali, usato in combinazione con un metodo di ottimizzazione come per esempio la discesa stocastica del gradiente, che vedremo più avanti. Iniziamo prendendo in esame, ancora una volta, la rete neurale più semplice che esista, composta da un unico neurone, che prende in ingresso un singolo input e produce un singolo output. Vogliamo calcolare il gradiente della funzione di perdita J rispetto ai pesi w_1 e w_2



Calcoliamo prima la derivata di J rispetto a w_2 , $\frac{\partial J(W)}{\partial w_2}$. Questa componente del gradiente indica quanto un piccolo cambiamento del peso w_2 possa influenzare la perdita della rete. Applicando la *regola della catena* (*chain rule*) risulta

$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial y} \cdot \frac{\partial y}{\partial w_2}$$

Supponendo ora di voler calcolare la derivata di J rispetto a w_1 , $\frac{\partial J(W)}{\partial w_1}$. Per trovare la prima componente del gradiente, andiamo ancora una volta ad applicare la *chain rule*, trovando

$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial y} \cdot \frac{\partial y}{\partial w_1}$$

da qui applichiamo ricorsivamente la *chain rule* sull'ultima componente, fintanto che questa si trovi in una forma che non si possa espandere ulteriormente. Nell'esempio in esame avremo solo due iterazioni, ma il discorso è valido per la struttura di una qualsiasi rete neurale. Tutte le singole componenti derivate possono essere propagate attraverso i cosiddetti *hidden layers* della rete neurale, fino ad arrivare a valutare ogni peso al suo interno. Da qui il nome *retropropagazione (backpropagation)*: il processo inizia dall'output e giunge fino all'input. Questo processo viene ripetuto molte volte durante la fase di addestramento, propagando informazioni al fine di "accordare" i pesi della rete per migliorarne le prestazioni e minimizzarne le perdite. Quanto appena descritto illustra l'algoritmo di retropropagazione, che rappresenta il cuore dell'addestramento delle reti neurali. In teoria si tratta di un processo molto semplice, trattandosi di semplici istanziazioni della *chain rule*, ma andremo nel seguito ad illustrare delle situazioni che potrebbero rendere l'addestramento di reti neurali in realtà estremamente complicato nella pratica.

2.5 Le Reti Neurali in Pratica : Ottimizzazione

Nella pratica, nell'ambito dell'ottimizzazione delle reti neurali, il cosiddetto *loss landscape*, ovvero la rappresentazione grafica della nostra funzione di perdita, è molto più vicina alla figura 2.7, piuttosto che a quella utilizzata nell'esempio precedente 2.6. Si tratta di un'illustrazione di un articolo uscito diversi anni fa ([07], figura 2.7), il cui obiettivo era quello di cercare di visualizzare il *landscape* di una rete neurale profonda. Risulta dunque chiaro quanto impegnativo possa essere, lavorando in un ambiente simile, riuscire a raggiungere l'obiettivo prefissato: trovare il minimo. Per affrontare le sfide cui ci pone davanti questo compito, riprendiamo per semplicità l'equazione di aggiornamento dei pesi definita nel *gradient descent* :

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$

Approfondiamo ora la funzione del parametro η . Come già accennato, tale parametro prende il nome di *tasso di apprendimento (learning rate)*, e determina essenzialmente di quanto ci dobbiamo spostare nella direzione opposta al gradiente trovato ad ogni singola iterazione della *back propagation*. Nella pratica, settare il *tasso di apprendimento* può essere particolarmente impegnativo. I progettisti di reti neurali hanno il compito di settare η , e scegliere un valore sbagliato può portare a importanti conseguenze. Se settassimo per esempio il *learning rate* su un valore troppo basso, la rete apprenderebbe molto lentamente.

Se η non è abbastanza elevato, non solo il sistema converge lentamente, talvolta la rete potrebbe non convergere affatto, rimanendo bloccata in un minimo locale. D'altra parte, se settassimo η ad un valore troppo elevato, ciò che potrebbe accadere è il verificarsi del fenomeno di *overshoot*, per il quale potremmo persino iniziare a divergere dalla soluzione. Il gradiente può esplodere, creando un modello instabile e rendendo l'addestramento della rete inconcludente. Nella realtà si cerca sempre di trovare il giusto mezzo, una soluzione che si pone a metà tra un settaggio troppo basso ed uno troppo elevato, in modo tale che il *learning rate* sia grande abbastanza da attraversare (*overshoot*) qualcuno dei minimi locali, ponendosi in una ragionevole porzione dello spazio di ricerca dove si possa effettivamente convergere alla soluzione ideale.

2.5.1 Learning Rate Ideale

Abbiamo due principali idee che guidano il settaggio del *tasso di apprendimento*:

1. l'idea alla base del primo approccio è molto semplice: si provano una vasta gamma di valori da assegnare η fino a che non si trova quello che funziona meglio: vengono addestrate diverse reti neurali parallelamente e le si osserva, cercando quella che raggiunge le prestazioni migliori. Si tratta di un metodo esaustivo.
2. una seconda, possibile strada da percorrere si basa sull'idea di progettare un algoritmo di *learning rate* che si possa effettivamente adattare alla nostra rete neurale e che sia in grado di adattarsi al suo *landscape*. Tendiamo a preferire questa seconda idea alla prima. Il che ci porta al concetto di *adaptive learning rates*

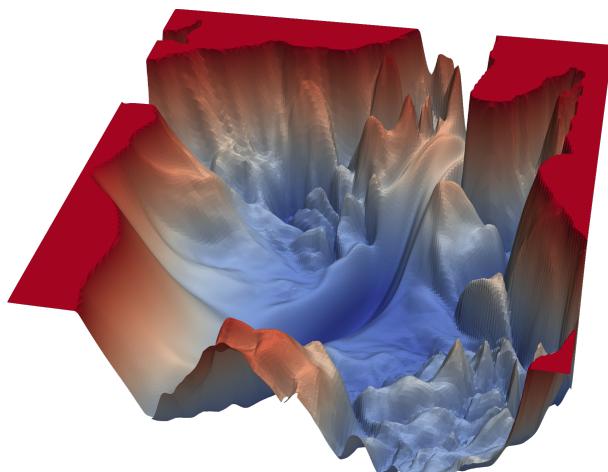


Figura 2.7: [07]

2.5.2 Adaptive Learning Rates

Il *learning rate* η , quindi la velocità a cui l'algoritmo si sta fidando dei gradienti che vede di iterazione in iterazione, dipenderà ora da diversi fattori riscontrati nel *learning landscape*: il valore del gradiente ad ogni specifica iterazione, quanto velocemente stiamo imparando, e da tante altre possibili opzioni che si possono settare come parte dell'addestramento di reti neurali.

2.5.3 Un esempio pratico

```

1 import tensorflow as tf
2 model = tf.keras.Sequential([...])
3 #pick a TensorFlow optimizer
4 optimizer = tf.keras.optimizer.SDG()
5 while True: #loop forever
6     #forward pass through the network
7     prediction = model(x)
8     with tf.GradientTape() as tape:
9         #compute the loss
10    loss = compute_loss(y, prediction)
11    #update the weights using the gradient
12    grads = tape.gradient(loss, model.trainable_variables)
13    optimizer.apply_gradients(zip(grads, model.trainable_variables))

```

Illustriamo per sommi capi il codice che tratta quanto discusso sino a questo momento, riservandoci di trattare l'aspetto pratico della questione in maniera più approfondita nel capitolo successivo (Capitolo 3). Per prima cosa definiamo il modello (linea 2); per ciascuna componente del modello avremo bisogno di un *optimizer* (linea 5), un ottimizzatore, di cui abbiamo appena parlato e che viene definito insieme al *learning rate* e definisce quanto velocemente si vuole ottimizzare il *loss landscape* su cui si lavora. All'interno del *loop* (linea 7 in giù), verranno analizzati tutti i campioni all'interno del nostro *training data set* ed essenzialmente si osserverà come migliorare la rete (grazie alla funzione discussa del gradiente) prima di effettuare, di fatto, tali migliorie, spostandosi in determinate direzioni tramite l'aggiornamento dei pesi. Questo processo viene ripetuto, fintanto che la rete neurale converge a qualche sorta di soluzione.

2.5.4 Mini Batches

Affrontiamo brevemente alcune tra le opzioni per l'addestramento delle reti neurali nella pratica. Focalizziamoci sull'idea del *batching* (*dosaggio*) dei dati in quelli che sono chiamati *mini batches*, che rappresentano lotti, porzioni più piccole dei nostri dati. Per farlo richiamiamo ancora una volta l'algoritmo del *gradient descent*. Il calcolo del gradiente,

di cui abbiamo parlato e di cui abbiamo sottolineato l'importanza, è in realtà un calcolo straordinariamente costoso in termini di capacità computazionali. Questo perché, come abbiamo visto, ciascuna derivata viene calcolata come il totale attraverso tutte le porzioni del *data set* di cui la rete dispone. Nella maggior parte dei problemi affrontati nella pratica, semplicemente non è possibile calcolare un gradiente su un intero *data set*, viste le dimensioni dei *data set* odierni. Vanno dunque cercate delle alternative. Anziché calcolare le derivate parziali dei gradienti sull'intero *data set*, il gradiente viene calcolato su un singolo campione del *data set*. Ovviamente la stima del gradiente sarà esattamente questo: una stima. Potrebbe grossomodo riflettere l'andamento dell'intero *data set*, ma dal momento che parliamo di un singolo campione, di fatto la possibilità che sia poco aderente a ciò che rappresenta la totalità dei dati è molto alta. Il vantaggio di un approccio simile è tuttavia sicuramente dato dalla velocità di calcolo, di molto maggiore. Computazionalmente dunque abbiamo dei vantaggi considerevoli, d'altra parte si tratta di un metodo estremamente stocastico. Non si parla più di *gradient descent*, ma di *stochastic gradient descent*. Come spesso accade si cerca anche in questa situazione una via di mezzo tra i due estremi, ed in questa situazione il *middle ground* sta nel calcolo del gradiente, anziché su un singolo campione nel *data set*, su un *lotto* di campioni, un *mini batch*. Ora, questi gradienti risulteranno ancora computazionalmente efficienti da calcolare dal momento lavoriamo su *mini batch*, non troppo grandi (si parla dell'ordine di decine o centinaia di campioni), e cosa più importante, avendo esteso quel singolo campione di partenza a qualcosa di ragionevolmente vicino ad un centinaio di campioni, la stocasticità risulta significativamente ridotta. L'accuratezza del modello ne trae beneficio. Ci troviamo di fronte ad un algoritmo molto più veloce da calcolare rispetto al *gradient descent* e molto più accurato ed aderente alla realtà nei risultati rispetto allo *stochastic gradient descent*. Questa miglioria nell'accuratezza del calcolo del gradiente permette alla rete di convergere ad una soluzione molto più velocemente di quanto sarebbe stato possibile in pratica secondo le limitazioni del *gradient descent*. Da qui potremmo anche aumentare il *learning rate* η , perché ci possiamo fidare maggiormente di ciascuno di questi gradienti: stiamo infatti calcolando la media su un *batch*, il che porterà a risultati di gran lunga superiori dal punto di vista delle prestazioni rispetto alla versione che segue il modello stocastico. Aumentando η , come abbiamo visto, diamo possibilità alla rete di apprendere più velocemente. Livelli ancora più significativi di accelerazione sull'intero problema possono essere raggiunti andando a parallelizzare computazionalmente in maniera massiva l'intero algoritmo, lavorando in parallelo su diversi *batch*.

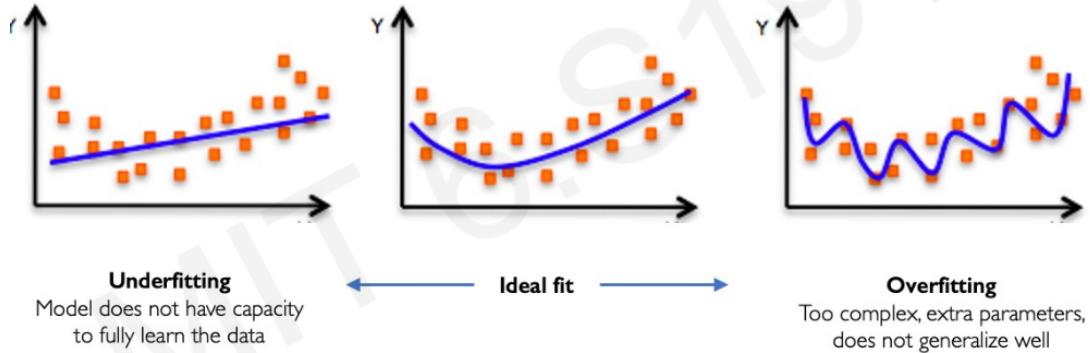


Figura 2.8: (da [06])

2.6 Overfitting

Il Machine Learning è la tecnologia dedicata all’individuazione di relazioni e regolarità nei dati. Permette di estrarre quelle di cui possiamo fidarci, evitando di essere ingannati da coincidenze, al fine di fare previsioni affidabili. Un fenomeno molto studiato è il cosiddetto *overfitting*, ovvero l’eccessiva flessibilità nello spiegare le osservazioni, incorporando anche relazioni accidentali. Negli ambiti di *statistica*, *machine learning* e *metodo scientifico* si è studiato a fondo questo rischio al fine di realizzare metodi rigorosi per evitarlo, oggi incorporati nel software che usiamo per addestrare le nostre macchine intelligenti. Quando ottimizziamo una rete neurale mediante l’utilizzo dello *stochastic gradient descent*, siamo posti davanti alla sfida dell’*overfitting*. Vogliamo costruire una rete neurale, o più in generale un modello di machine learning che sia in grado di descrivere accuratamente qualche pattern presente nei nostri dati. È importante ricordare che in ultima istanza non vogliamo descrivere i pattern presenti nei nostri *training data*, che rendono possibile l’addestramento, ma idealmente vorremmo definire dei pattern che si presentano nei *test data*, dati mai visti prima. Ovviamente non osserviamo i *test data*, ma solamente i *training data*, dunque ci si pone davanti un nuovo, fondamentale problema: estrarre pattern dai *training data* sperando che questi possano essere generalizzati sui nostri *test data*. Detto con altre parole: vogliamo costruire dei modelli che possano imparare delle rappresentazioni dai nostri *training data*, rappresentazioni che possano essere generalizzate anche quando mostriamo a questi modelli dei dati mai visti prima. Assumiamo di voler costruire una linea che possa descrivere o trovare dei pattern nei punti rappresentati in figura 2.8. Se si ha a disposizione una rete neurale molto semplice, rappresentabile con una linea retta, è possibile fornire una descrizione di questi dati al più subottimale, poiché dal momento che i dati sono non lineari, si andranno a perdere tutte le sfumature e le sottigliezza presenti all’interno del *data set*. Spostandosi sulla destra della figura si osserva

una situazione completamente capovolta: ci troviamo di fronte ad un modello più complesso, ma in realtà *over expressive*, che sta catturando anche le più piccole sfumature, le spurious presenti nei nostri *training data*, che non si riveleranno poi in realtà rappresentative nell'analisi dei nostri *test data*. Idealmente, ancora una volta, ci si vuole porre nel mezzo. Siamo alla ricerca di un modello simile a quello rappresentato nel centro della figura, che raffiguri un po' la terra di mezzo in questa situazione: né troppo complesso né troppo semplice, raggiunge buone performance quando gli vengono sottoposti nuovi dati.

2.6.1 Regolarizzazione

La *regolarizzazione (regularization)* è una tecnica che si può introdurre nella *training pipeline* per scoraggiare l'apprendimento di modelli eccessivamente complessi. Si tratta di un argomento critico: le reti neurali, come spesso sottolineato, sono modelli estremamente larghi e pertanto estremamente inclini all'*overfitting*. Disporre di tecniche per la regolarizzazione ha delle estreme implicazioni per il successo delle reti neurali, particolarmente nella loro capacità di generalizzare correttamente, al di là quindi dei *training data*, i nostri domini di *testing*. Osserviamo di seguito qualche tecnica di regolarizzazione :

- *Dropout*: è la tecnica più popolare per regolarizzare i dati nell'ambito del deep learning. L'idea su cui pone le sue fondamenta è in realtà molto semplice: durante l'addestramento viene selezionato in maniera aleatoria qualche *subset*, sottoinsieme di neuroni della rete neurale, che verranno spenti a ciascuna iterazione con una qualche probabilità, anche questa casuale. Dunque, con qualche probabilità, il set di neuroni selezionati sarà spento e acceso su diverse iterazioni durante il processo di addestramento. Questo processo essenzialmente forza la rete neurale ad apprendere un insieme di modelli differenti. Ad ogni iterazione la rete sarà esposta ad un modello internamente leggermente differente rispetto a quelli che lo avevano preceduto, alle iterazioni precedenti. La rete deve dunque imparare come costruire diversi percorsi interni per processare la stessa informazione, non potendo realmente fare completo affidamento su quanto imparato nelle precedenti iterazioni. Il modello è forzato a catturare un qualche significato più profondo all'interno dei percorsi e questo può rivelarsi uno strumento estremamente potente. In prima istanza viene infatti abbassata la capacità della rete in maniera significativa (se si pensa che la probabilità di *on/off* di un certo neurone ad una certa iterazione sia pari a 0.5, abbassiamo la capacità circa del 50%),

```

1   tf.keras.layers.Dropout(p=0.5)
2

```

va poi considerata la maggiore facilità nel processo di addestramento: il numero di pesi che hanno gradienti in questo caso vengono considerevolmente ridotti. In



Figura 2.9: Early Stop graph (da [06])

sostanza ad ogni iterazione un set di neuroni scelti casualmente viene ignorato ed il comportamento della rete che ne deriva permette una migliore generalizzazione dei dati.

- *Early Stopping*: si tratta di un metodo di regolarizzazione molto ampio, che va ben al di là dell'utilizzo nelle reti neurali. Sappiamo che la definizione di *overfitting* rispecchia una situazione in cui il nostro modello inizia a rappresentare i *training data* più di quanto rappresenti i *testing data*. Alla base è questo ciò a cui si riduce l'*overfitting*. Se mettiamo da parte dei *training data*, in maniera tale da poterli usare separatamente, non usandoli per addestrare la rete, ma facendogli fare le veci dei *testing data*, al punto che potremmo pensarli come *testing data sintetici*, possiamo monitorare come la nostra rete sta imparando, servendo questi dati mai visti prima dalla rete e tenendo traccia del percorso di apprendimento della stessa sia sul *training set* che su quello che chiameremo *testing set sintetico*. Mentre la rete viene addestrata vedremo diminuire in entrambi i set di dati le perdite, ma ci sarà un punto in cui la differenza tra i due valori di perdita inizierà ad aumentare (aumenta la perdita del *testing set sintetico*) ed è quello l'esatto punto in cui inizia la fase di *overfitting* (*overfit* sui *training data*). Questo pattern continua per il resto dell'addestramento, diventa dunque di fondamentale importanza individuare quel punto, in cui le due performance, osservate separatamente, iniziano a divergere. Assumendo che il *test set sintetico* sia rappresentativo dei veri *test set*, quello è infatti il punto oltre il quale l'accuratezza del modello andrà esclusivamente a peggiorare. È proprio lì (vedere figura 2.9) che vorremo effettuare il cosiddetto *early stop*, in modo tale da regolarizzare la performance su un punto ottimale. Scegliere accuratamente il punto in cui fermarsi è importante: fermarsi prima di quel punto produrrebbe un *underfit model*, quando avremmo potuto avere un modello migliore cui sottoporre i

testing data. Come sempre bisogna andare alla ricerca del giusto compromesso: non ci si può fermare troppo tardi ma nemmeno troppo presto.

Capitolo 3

Costruzione di un Modello

3.1 Reti Neurali: esempi pratici

Per passare all'effettiva realizzazione del progetto prendiamo prima familiarità con l'ambiente di lavoro che verrà utilizzato, osservando nella pratica qualche esempio facilmente realizzabile grazie all'apporto di *library* come le già citate e brevemente osservate *tensorflow* e *keras*. Seguiremo liberamente alcuni tra gli esempi esposti su *Google Developers* ([08]) nella sezione *AI & Machine Learning*. Passiamo così alla parte pratica del lavoro, mettendo insieme quanto visto sino ad ora per realizzare un primo modello di *machine learning*, con riferimento al codice sotto, che sia in grado di apprendere la relazione che lega il vettore x al vettore y .

```

1 import tensorflow as tf
2 from tensorflow import keras
3
4 model = keras.Sequential([
5     keras.layers.Dense(units=1, input_shape=[1])
6 ])
7 model.compile(
8     optimizer='sgd',
9     loss='mean_squared_error'
10 )
11
12 x = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
13 y = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)
14
15 model.fit(x, y, epochs=500)
16 print(model.predict[10.0])

```

Alla linea 4 definiamo il *modello*. Un modello è una rete neurale addestrata e qui ci troviamo di fronte alla rete neurale più semplice che si possa costruire: è dotata infatti

di un unico *layer* costituito da un singolo neurone, cui viene dato in input un singolo valore (x). Per ciascuna componente del modello avremo bisogno di un *optimizer*, un ottimizzatore, di cui abbiamo parlato al capitolo precedente e che viene definito insieme al *learning rate* e definisce in che modo e quanto velocemente si vuole ottimizzare il *loss landscape* su cui si lavora. Qui l’obiettivo è quello di fare in modo che la rete neurale predica l’ y per una specifica x . Alla linea 5 compiliamo il modello e definiamo le due funzioni (*loss* e *optimizer*) che rappresentano la chiave del *machine learning*. Il modello farà una predizione (*guess*) sulle relazione che lega x ed y . Durante l’addestramento, come visto, determinerà quanto veritiera sia effettivamente quella predizione (grazie all’utilizzo della *loss function*) per poi utilizzare l’*optimizer function* al fine di compierne una nuova. Logica vuole che la combinazione di queste due funzioni porti via via il modello sempre più vicino alla corretta relazione che lega i valori. L’addestramento vero e proprio inizia utilizzando il metodo *fit* (linea 10). Nel caso in esempio il processo sopra descritto verrà ripetuto per 500 volte ($epochs = 500$, dove $epochs$ indica il numero di volte in cui il modello lavora sull’intero *training set*). Compiuto questo processo disponiamo di un modello addestrato. Sarà dunque possibile predire un valore y , dato un nuovo x (linea 12, $x = 10.0$)

Vediamo ora come costruire una rete neurale al fine di riconoscere gli articoli di abbigliamento da un set di dati comune, chiamato *Fashion MNIST*. Contiene 70.000 immagini raffiguranti capi di abbigliamento in 10 diverse categorie. Ogni immagine, in scala di grigi (*grayscale*), ha dimensione 28×28 . In un’immagine realizzata in ”*scala di grigi*”, la quantità di informazioni fornite per ogni pixel è inferiore rispetto a quella necessaria in un’immagine a colori (*RGB*). Un colore grigio è infatti un colore in cui le componenti rosso, verde e blu hanno uguale intensità nello spazio RGB. Ogni pixel, lavorando con immagini *grayscale*, è rappresentato da un numero intero compreso tra 0 e 255. Quando si addestra una rete neurale è tuttavia più facile trattare tutti i valori tra 0 e 1, per questo si applichiamo un processo che prende il nome di *normalizzazione* (linee 7 – 8). Le etichette (*labels*) sono numeri che indicano la classe cui appartiene un determinato oggetto e sono fondamentali in fase di apprendimento. Lavoriamo su due differenti set di dati: *training set* e *testing set*. L’idea è quella di avere un insieme di dati da usare esclusivamente in fase di addestramento e un altro, più piccolo, da utilizzare dopo aver chiamato il metodo *fit*, al fine di controllare le effettive performance della rete. Lo scopo primo è infatti quello di portare il modello ad un livello tale da poter essere utilizzato per classificare dati mai visti prima. Il modello qui è sicuramente più complesso rispetto al precedente: il primo *layer* ha un input di dimensioni $28 \cdot 28$ (linea 11), coincidente con la dimensione delle immagini presenti nel *data set*, che verranno analizzate dal modello. L’ultimo *layer* ha invece dimensione 10: sono infatti in questo esempio 10 le diverse categorie di oggetti forniti da *mnist*, rappresentanti l’intero *dataset*. La rete prende in ingresso un insieme di

28 · 28 pixel ed offre in uscita uno di dieci possibili valori. Il numero 128 (linea 12) sta ad indicare che disporremo di 128 funzioni in corrispondenza del rispettivo *layer*, ciascuna delle quali presenta al proprio interno quei parametri di cui abbiamo tanto parlato: i *pesi* (*weights*). Definiamo anche qui, attraverso il metodo *compile*, *loss* ed *optimizer* per il nuovo modello (linea 17 – 18). La rete neurale verrà inizializzata con valori casuali, come già visto. La *loss function* misurerà quanto ottimale sia il risultato ottenuto da questa prima inizializzazione e poi, grazie all'*optimizer*, genererà nuovi parametri per le funzioni al fine di migliorarsi. *Softmax* (linea 13) estrae il più grande tra un *set* di valori, setta quel valore a 1 ed azzera tutti gli altri. Giunta all'ultimo *layer* la rete si limiterà quindi a trovare quel valore unitario per restituirlo. Al metodo *fit* (linea 22), vengono passate *training images* e *training lables*, perché il modello possa imparare ad accoppiarle correttamente; attraverso l'utilizzo del metodo *evaluate* (linea 24) vengono poi utilizzate le immagini che il modello non ha visto (*test set*), al fine di testarne le prestazioni. Si può infine (linea 25) ottenere predizioni per nuove immagini, chiamando il metodo *predict*. La limitazione di quanto appena visto sta nell'aver addestrato il modello su immagini tutte dello stesso tipo (28 · 28 pixel, con i soggetti centrati nelle immagini). Introduciamo quindi nel seguito il concetto di *Rete Neurale Convoluzionale*.

```

1 import tensorflow as tf
2 from tensorflow import keras
3
4 fashion_mnist = keras.datasets.fashion_mnist
5 (train_images, train_labels), (test_images, test_labels) = fashion_mnist
6           .load_data()
7
8 training_images = training_images / 255.0
9 test_images = test_images / 255.0
10
11 model = keras.Sequential([
12     keras.layers.Flatten(input_shape=(28, 28)),
13     keras.layers.Dense(128, activation=tf.nn.relu),
14     keras.layers.Dense(10, activation=tf.nn.softmax)
15 ])
16
17 model.compile(
18     optimizer = tf.keras.optimizers.Adam(),
19     loss = 'sparse_categorical_crossentropy',
20     metrics=['accuracy']
21 )
22
23 model.fit(training_images, training_labels, epochs=5)
24 test_loss, test_acc = model.evaluate(test_images, test_labels)
```

```
25 predictions = model.predict(test_images)
```

3.2 Reti Neurali Convoluzionali

L'idea alla base di una *rete neurale convoluzionale* è quella di filtrare le immagini prima di addestrare la rete. Una *convoluzione* è un *filtro* che viene applicato ad un'immagine al fine di evidenziarne le caratteristiche (*features*). Un *filtro* viene realizzato attraverso un insieme di moltiplicatori che svolgono il seguente compito: elaborano l'immagine marcandone le *features* caratterizzanti e sfocano quelle marginali e superflue. Tale processo prende il nome di *feature mapping* ed è illustrato in figura 3.1 (da [09]). Ogni pixel dell'immagine originaria verrà analizzato e da quest'analisi (che tiene conto dei pixel che lo circondano) verrà ricavato un nuovo valore da assegnargli nell'immagine risultante dal processo. Se si guarda, come in esempio, al singolo pixel dal valore 192 e si considera il filtro rappresentato dai valori evidenziati in rosso, basterà moltiplicare 192 per 4.5, ed ogni pixel adiacente per il corrispettivo valore all'interno del *filtro*, sommando poi ciascun risultato ottenuto per ricavare il valore risultante. Vediamo in figura 3.2 (da [09]) a cosa può portare l'applicazione di un particolare *filtro*: nel primo caso dall'immagine originale viene eliminato quasi tutto, ciò che è evidenziato sono le linee verticali; vale lo stesso discorso per il secondo caso, in cui ad essere evidenziate sono invece le linee orizzontali, grazie all'applicazione del differente filtro. Di seguito creiamo una *matrice di convoluzione*.

```
1 filter = [ [-1, -2, -1], [0, 0, 0], [1, 2, 1]]
```

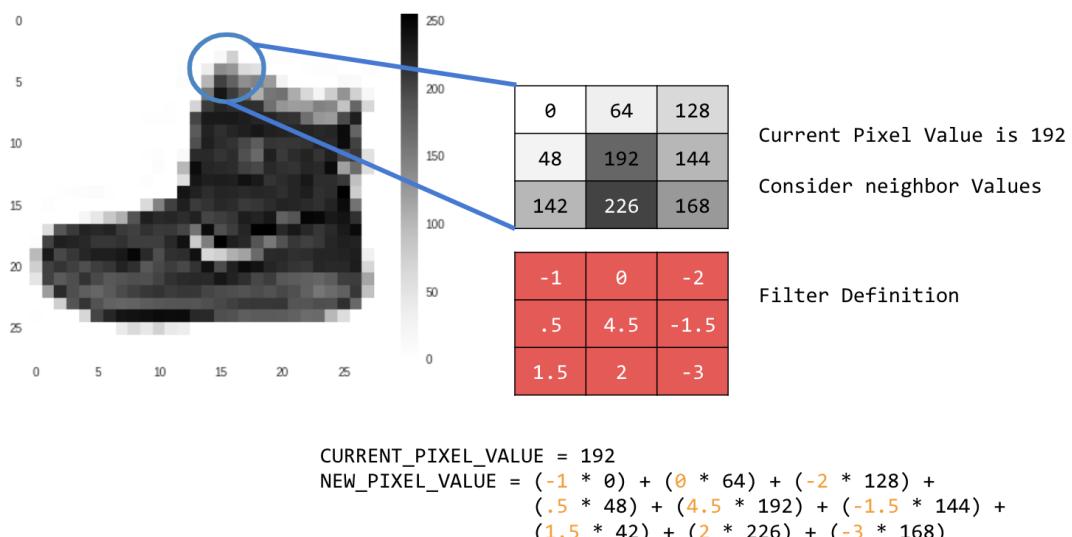


Figura 3.1: applicazione di un filtro

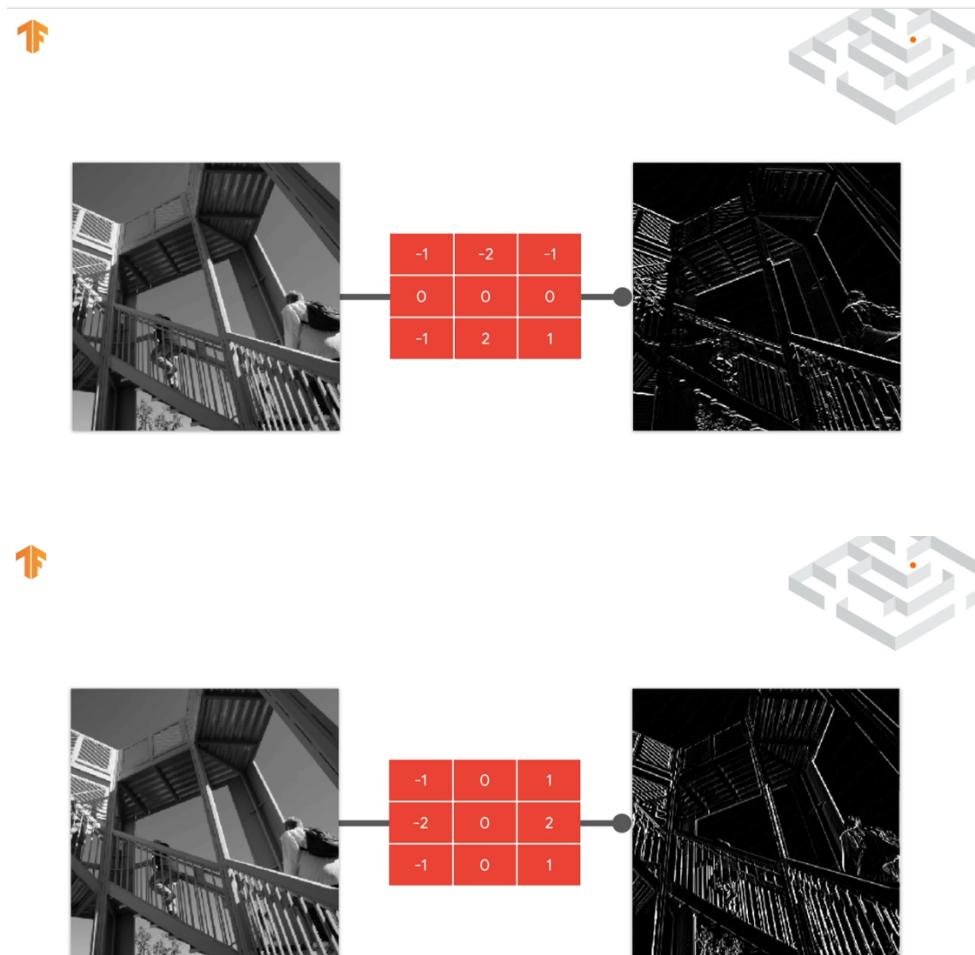


Figura 3.2: esempio di filtri

Calcoliamo ora i nuovi valori da assegnare a ciascun pixel, considerando (*sizex*, *sizey*) le dimensioni dell'immagine da elaborare

```

1 for x in range(1, sizex - 1):
2     for y in range(1, sizey - 1):
3         output_pixel = 0.0
4         output_pixel = output_pixel + (i[x - 1, y-1] * filter[0][0])
5         output_pixel = output_pixel + (i[x, y-1] * filter[0][1])
6         output_pixel = output_pixel + (i[x + 1, y-1] * filter[0][2])
7         output_pixel = output_pixel + (i[x-1, y] * filter[1][0])
8         output_pixel = output_pixel + (i[x, y] * filter[1][1])
9         output_pixel = output_pixel + (i[x+1, y] * filter[1][2])
10        output_pixel = output_pixel + (i[x-1, y+1] * filter[2][0])
11        output_pixel = output_pixel + (i[x, y+1] * filter[2][1])
12        output_pixel = output_pixel + (i[x+1, y+1] * filter[2][2])
13        output_pixel = output_pixel * weight

```

```

14 if(output_pixel<0):
15 output_pixel=0
16 if(output_pixel>255):
17 output_pixel=255
18 i_transformed[x, y] = output_pixel

```

Una volta definite le *features* caratterizzanti dell’immagine, è possibile sottoporre la risultante *feature map* ad un ulteriore operazione che prende il nome di *pooling*. Tale processo che consiste nel raggruppare i pixel dell’immagine per poi filtrarli in un sottoinsieme (*subset*): riduce la quantità di informazioni dell’immagine mantenendone le caratteristiche. Ci sono diverse modalità di *pooling*, noi (sotto) vediamo l’applicazione del cosiddetto *Max Pooling*, che consiste nell’iterare sull’immagine e, ad ogni punto, considerare una cella di 2 per 2 pixel, prendere il più grande tra i 4 e caricarlo nella nuova immagine.

```

1 new_x = int(size_x/2)
2 new_y = int(size_y/2)
3 newImage = np.zeros((new_x, new_y))
4 for x in range(0, size_x, 2):
5 for y in range(0, size_y, 2):
6 pixels = []
7 pixels.append(i_transformed[x, y])
8 pixels.append(i_transformed[x+1, y])
9 pixels.append(i_transformed[x, y+1])
10 pixels.append(i_transformed[x+1, y+1])
11 pixels.sort(reverse=True)
12 newImage[int(x/2), int(y/2)] = pixels[0]

```

Quindi per esempio effettuando un *pooling* come in linea 3 (codice sotto), ciascuna immagine verrà suddivisa in *set* di $2 \cdot 2$ pixel, da cui verranno estratti quelli aventi valore più alto. L’immagine viene ridotta ad un quarto della sua iniziale dimensione, ma le caratteristiche persistono e risaltano in primo piano. Questi *filtri*, di cui abbiamo iniziato a parlare con l’introduzione delle *reti convoluzionali*, non sono altro che parametri, in realtà molto vicini ai *pesi* delle reti neurali e, proprio come avviene per i *pesi*, il loro valore ottimale viene settato dal modello grazie al processo di apprendimento automatico: quando un’immagine viene passata al *convolutional layer* un numero di filtri, inizializzati in maniera casuale, viene applicato all’immagine. Il risultato di questa applicazione viene passato al *layer* successivo e la rete neurale compie un’operazione di *matching*. Col tempo, i filtri che danno luogo ad output dell’immagine che risultano nei migliori *match* vengono appresi: si parla di *feature extraction*. Una rete neurale del genere viene costruita, per esempio, attraverso il codice che segue

```

1 model = tf.keras.models.Sequential([
2     tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(28,
28, 1)),

```

```
3 tf.keras.layers.MaxPooling2D(2,2),  
4 tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(28,  
28, 1)),  
5 tf.keras.layers.MaxPooling2D(2,2),  
6 tf.keras.layers.Flatten(),  
7 tf.keras.layers.Dense(128, activation=tf.nn.relu),  
8 tf.keras.layers.Dense(10, activation=tf.nn.softmax)  
9 ])
```

Alla riga 2 specifichiamo l'*input shape* e facciamo in modo che la rete generi 64 filtri. Ciascuno di essi verrà moltiplicato attraverso l'immagine. Ad ogni *epoch* la rete vedrà quali filtri hanno prodotto i risultati migliori nel processo di *matching* tra le immagini e le rispettive *labels*, seguendo lo stesso principio che l'aveva portata ad "accordare" i *pesi* nel *dense layer*. Il *max pooling* per comprimere l'immagine ed evidenziarne le caratteristiche è alla riga successiva del codice. È poi possibile impilare più *convolutional layers* gli uni sugli altri per scomporre l'immagine e provare ad apprendere da *features* sempre più astratte. Seguendo la metodologia appena illustrata, la rete inizia ad apprendere basandosi sulle caratteristiche dell'immagine anziché su *raw patterns*, ovvero modelli grezzi, dei pixel.

Capitolo 4

Raccolta Dati

4.1 Strumentazione

4.1.1 Misuratore di pressione - RS7 Intelli IT

Per le misurazioni della pressione si è fatto uso del misuratore di pressione da polso *RS7 Intelli IT* (per specifiche e documentazione si rimanda a [03]). Lo *sfigmomanometro digitale* è composto da un manicotto e da un piccolo apparecchio con schermo digitale. Per la misurazione si inserisce il braccio nel manico, si allaccia e, una volta avviato, raggiunta la posizione ottimale alla misurazione, l'apparecchio procede a gonfiare la camera d'aria, a rilasciarla e a mostrare sul display i valori corrispondenti alla *massima (sistolica)* e alla *minima (diastolica)*. Tali risultati vengono poi inviati ad un dispositivo associato tramite Bluetooth, facilitando la raccolta dati. Il sensore di posizionamento indica quando il bracciale è indossato nella posizione corretta per poter eseguire una misurazione. Mentre misura la pressione arteriosa e la frequenza delle pulsazioni, il dispositivo indica l'eventuale rilevamento di un battito cardiaco irregolare o di ipertensione.

4.1.2 Sensore GSR - Shimmer3 GSR+

Il sensore *Shimmer3 GSR+* (per specifiche e documentazione si rimanda a [04]) è stato utilizzato per effettuare misurazioni della *risposta galvanica cutanea* (GSR). Lo *Shimmer3 GSR+* permette di misurare la resistenza elettrodermica posizionando due elettrodi su due dita di una mano ed applicando un potenziale elettrico tra i due punti di contatto sulla cute al fine di rilevare il flusso di corrente risultante tra di essi. L'attività conduttrice della pelle è dovuta all'apertura delle ghiandole sudoripare in risposta a stimoli interni ed esterni: il livello di umidità sulla pelle aumenta e permette alla corrente elettrica di fluire più facilmente, modificando l'equilibrio fra ioni positivi e negativi nel fluido secreto (diminuisce la resistenza cutanea e dunque aumenta la conduttanza, suo reciproco).



Figura 4.1: RS7 Intelli IT

Oltre che al segnale GSR, grazie ad un ulteriore dispositivo indossabile da applicare ad un dito o al lobo di un orecchio, il sensore permette parallelamente la misura di un segnale *fotopletismografico* (PPG) che, con una certa elaborazione, può essere utilizzato per stimare il polso o la frequenza cardiaca di un soggetto.

4.2 Misurazioni

Vogliamo sviluppare un metodo non invasivo, *camera based*, basato quindi sull'utilizzo di foto che vengono utilizzate nell'ambito della *Computer Vision*. Per ciascuna misurazione abbiamo raccolto immagini che inquadrano: dorso della mano, polso e interno del gomito(braccio). L'obiettivo ultimo, ricordiamo, è quello di addestrare una rete a comprendere la correlazione tra immagini e valori misurati. Ciascun soggetto che si è prestato



Figura 4.2: Shimmer3 GSR+

al processo di raccolta dati è stato sottoposto ad un totale di nove misurazioni (cui sono associate 27 immagini: un terzo per il dorso della mano, un terzo relative al polso e le rimanenti aventi come soggetto il braccio). Questo al fine di ampliare quanto più possibile il numero di dati su cui poter lavorare. Come più volte affermato operiamo infatti sotto un approccio *data driven*, dove sono proprio i dati di cui disponiamo a guidare il comportamento dei nostri modelli. A fronte del medesimo numero di dati sarebbe chiaramente ottimale disporre di informazioni raccolte su quanti più soggetti diversi possibile. Disponendo però di un numero limitato di persone su cui poter effettuare misurazioni, si è fatta la scelta di sottoporre quegli stessi soggetti ad un numero di misurazioni tale da poter raggiungere un set di dati sufficientemente grande sul quale addestrare il modello. Tre delle nove misurazioni sono state effettuate su ciascun soggetto a riposo; le seguenti tre a seguito di uno sforzo; le rimanenti in una condizione idealmente a metà tra le prime due.

Capitolo 5

Modello per la stima della Pressione Arteriosa

5.1 Costruzione del database

Illustriamo nel seguito il processo di costruzione del database. A tal fine facciamo riferimento alla figura 5.1 per visualizzare la struttura sulla quale si è iniziato a lavorare. Nella *directory* principale del progetto, in figura sotto il nome *main*, sono stati inseriti tutti i dati raccolti (immagini e relative misurazioni) in una struttura ad albero la cui radice coincide con *main/data/*. Ogni percorso sotto tale *directory*, indipendentemente da quanto si trova nel mezzo -che non è altro che il risultato della suddivisione delle misurazioni in base a giorni e soggetti- si conclude sempre con tre cartelle (*Braccio*, *Dorso* e *Polso*), che permettono di valutare ciascun set di immagini, relative alla stessa porzione fotografata al momento della misurazione, indipendentemente dagli altri due. A partire da questa iniziale struttura lanciamo i due script MATLAB riportati di seguito (*buildDataset.m* e *discretizeTargets.m*) al fine di suddividere le varie misurazioni e le relative immagini per catalogarle secondo valori di pressione bassi, normali e alti. Concentriamoci sul primo script, *main/script/buildDataset.m*, riportato di seguito:

buildDataset.m

```

1      %% Build dataset
2      clc;
3      clear;
4      close all;
5
6
7      data_path = '../data';
8      dirs = dir(data_path);

```

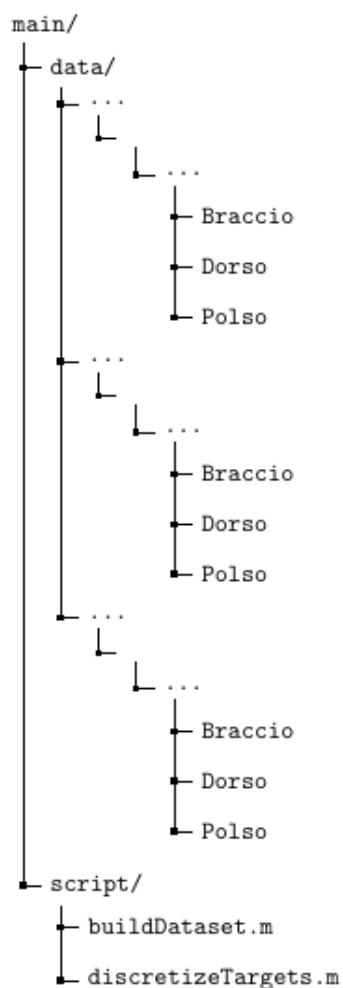


Figura 5.1

```

9
10    if ~exist('..../dataset', 'dir')
11        mkdir('..../dataset')
12    end
13
14
15    %%
16    targets = [];
17    counter = 1;
18
19    for i = 3 : numel(dirs)
20        dirs_1 = dir(strcat(data_path, '/', dirs(i).name));
21
22        for j = 3 : numel(dirs_1)
23            dirs_2 = dir(strcat(data_path, '/', dirs(i).name, '/', dirs_1(j).
24                .name));
25
26            for k = 3 : numel(dirs_2)
27                targets_path = dir(strcat(data_path, '/', dirs(i).name, '/',
28                    dirs_1(j).name, '/', dirs_2(k).name, '*.csv'));
29                targets_file = readtable(strcat(targets_path.folder, '/',
30                    targets_path.name), 'PreserveVariableNames', true);
31                dirs_3 = dir(strcat(data_path, '/', dirs(i).name, '/', dirs_1(j).
32                    .name, '/', dirs_2(k).name, '/Dati'));
33
34                if i == 3
35                    T = removevars(targets_file, {'Data_misurazione', 'Fuso_orario', 'Pulse', 'Nome_modello_dispositivo'});
36                else if i == 5
37                    T = removevars(targets_file, {'Eta'});
38                else
39                    T = targets_file;
40                end
41            end
42            targets = [targets; T];
43
44            for z = 3 : numel(dirs_3)
45                dirs_4 = dir(strcat(data_path, '/', dirs(i).name, '/', dirs_1(j).
46                    .name, '/', dirs_2(k).name, '/Dati/', dirs_3(z).name));
47
48                type = erase({dirs_4.name}, '');
49                if strcmp(type{n,:}, 'Braccio') || strcmp(type{n, :}, 'Polso')
50                    || strcmp(type{n, :}, 'Dorso')
51                        if ~exist(strcat('..../dataset/imgs/', type{n, :}), 'dir')
52                            mkdir(strcat('..../dataset/imgs/', type{n, :}));

```

```

48      end
49
50      imgs_path = dir(strcat(data_path, '/', dirs(i).name, '/', dirs_1
51          (j).name, '/', dirs_2(k).name, '/Dati/', dirs_3(z).name, '/',
52          type{n, :}, '/*.jpg'));
53      %disp(imgs_path);
54      img = imread(strcat(imgs_path.folder, '/', imgs_path.name));
55
56      % Extract green channel
57      bw = imbinarize(img(:, :, 2), 'adaptive', 'Sensitivity', 0.75);
58
59      img(:, :, 1) = double(img(:, :, 1)) .* bw;
60      img(:, :, 2) = double(img(:, :, 2)) .* bw;
61      img(:, :, 3) = double(img(:, :, 3)) .* bw;
62
63      lab = rgb2lab(img);
64      light = lab(:, :, 1);
65
66      scale_light = 1 - (light / 100.0);
67
68      lab(:, :, 1) = lab(:, :, 1) .* scale_light;
69      img = lab2rgb(lab, 'OutputType', 'uint8');
70      img(:, :, 1) = img(:, :, 2);
71      img(:, :, 3) = img(:, :, 2);
72
73      crop = min(size(img, 1), size(img, 2));
74      r = centerCropWindow2d(size(img), [crop crop]);
75      img = imcrop(img, r);
76      img = imresize(img, [512 512]);
77
78      imwrite(img, strcat('../dataset/imgs/', type{n, :}, '/',
79          num2str(counter), '.jpg'));
80
81      end
82
83      continue;
84
85      end
86      writetable(targets, '../dataset/targets.csv');

```

Attraverso *buildDataset.m* andiamo a costruire una nuova cartella, *main/dataset/*, la cui struttura a seguito dell'esecuzione dello script è raffigurata in figura 5.2: tale *directory* conterrà le immagini, elaborate e suddivise a seconda del soggetto che raffigurano (*Braccio*,

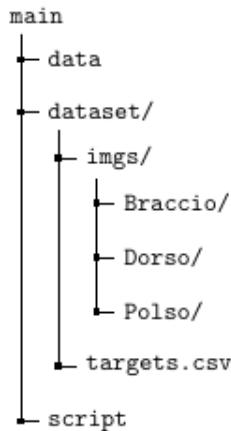


Figura 5.2

Dorso, Polso). Nel file *targets.csv* sono stati invece riportati tutti i valori relativi ai due indici di nostro interesse (problema gestito nel codice sopra alle linee 30 – 38): *pressione arteriosa sistolica* (SYS) e *pressione arteriosa diastolica* (DIA). A ciascuna immagine (i cui nomi sono indicati numericamente da 1 a 456) corrisponde una riga della tabella in *targets.csv*, individuata dall’indice coincidente al proprio nome. Vediamo ora di seguito il secondo script MATLAB, *main/script/discretizeTargets.m*

discretizeTargets.m

```

1      %% 
2      clc;
3      clear;
4      close all;
5
6
7      %% Load images and targets
8      type = 'Polso';
9      data_path = strcat('../dataset/imgs/', type);
10     dirs = dir(data_path);
11
12     targets = readtable('../dataset/targets.csv', '
13         PreserveVariableNames', true);
14
15     if ~exist(strcat('../dataset/classes/', type, '/sys/low'), 'dir')
16         mkdir(strcat('../dataset/classes/', type, '/sys/low'))
17         mkdir(strcat('../dataset/classes/', type, '/sys/normal'))
18         mkdir(strcat('../dataset/classes/', type, '/sys/high'))
19     end
  
```

```
19
20     if ~exist(strcat('../dataset/classes/', type, '/dia/low'), 'dir')
21         )
22     mkdir(strcat('../dataset/classes/', type, '/dia/low'))
23     mkdir(strcat('../dataset/classes/', type, '/dia/normal'))
24     mkdir(strcat('../dataset/classes/', type, '/dia/high'))
25     end
26
27     for i = 1 : size(targets, 1)
28
29         img = imread(strcat('../dataset/imgs/', type, '/', num2str(i), '.jpg'));
30
31         sys = targets{i, 1};
32         dia = targets{i, 2};
33
34         if sys < 90
35             sys_disc = 'low';
36         elseif sys >= 90 && sys <= 120
37             sys_disc = 'normal';
38         elseif sys > 120
39             sys_disc = 'high';
40         end
41
42         path = strcat('../dataset/classes/', type, '/sys/', sys_disc, '/',
43             ', num2str(i), '.jpg');
44         imwrite(img, path);
45
46         if dia < 60
47             dia_disc = 'low';
48         elseif dia >= 60 && dia <= 90
49             dia_disc = 'normal';
50         elseif dia > 90
51             dia_disc = 'high';
52         end
53
54         path = strcat('../dataset/classes/', type, '/dia/', dia_disc, '/',
55             ', num2str(i), '.jpg');
56         imwrite(img, path);
57     end
```

Finita l'esecuzione, lo script porterà alla creazione di un'ulteriore cartella (*main/dataset/classes*), portando a fine la realizzazione di *main/dataset*, la cui struttura completa è riportata in figura 5.3. Si è iniziato facendo uso delle sole immagini. Dagli esperimenti compiuti le più significative sono risultate essere quelle aventi per soggetto il polso. Ci

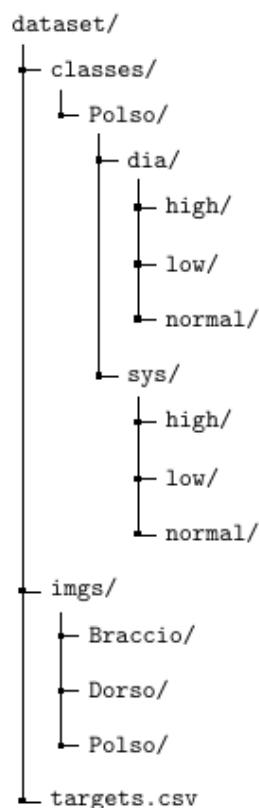


Figura 5.3

class	SYS	DIA
high	$sys > 120$	$dia > 90$
normal	$90 \leq sys \leq 120$	$60 \leq dia \leq 90$
low	$sys < 90$	$dia < 60$

Tabella 5.1: suddivisione per classi (*high*, *normal*, *low*)

siamo per questo concentrati sullo studio delle ultime (linea 8, *discretizeTarget.m*), suddividendo *diastolica* e *sistolica* (*dia* e *sys*) per classi individuate da valori alti, bassi e normali (*high/*, *low/*, *normal/*) per entrambi gli indici, secondo i valori in tabella 5.1, riferiti alle linee di codice 34 – 40 (*sys*) e 45 – 51 (*dia*)

5.2 Sviluppo della Rete Neurale

5.3 Pressione Arteriosa Sistolica

Abbiamo trattato separatamente, seppur in maniera equivalente, i problemi di classificazione di immagini per valori di *pressione sistolica* e *diastolica*. Approfondiamo nel seguito la trattazione relativa alla classificazione dei primi, per i quali disponiamo di un *dataset* fortemente più bilanciato rispetto al *set* relativo ai campioni derivanti dalle misurazioni dei valori *minimi* della pressione. A partire dalla struttura delle cartelle, ottenuta e illustrata nella sezione precedente, dividiamo il *dataset* a nostra disposizione in maniera tale da poter disporre di un *training set* e di un *validation set*, ottenendo come risultato quanto illustrato in figura 5.4 (ignoriamo qui il percorso *classes/Polso/dia*, seppur presente).

Addestramento a partire da un piccolo *dataset*

Il nostro obiettivo è quello di costruire un modello di classificazione d'immagini a partire dal *dataset* ottenuto come risultato delle misurazioni effettuate, piuttosto ridotto. Disponiamo infatti di poco meno di 500 campioni. L'addestramento su un insieme di tali dimensioni, vedremo, produrrà comunque risultati ragionevoli, nonostante una relativa mancanza di dati. Illustriamo, nel seguito, come il problema è stato affrontato e quali comuni strategie sono state applicate a tal fine. Come detto, a guidare i processi di *deep learning*, è sempre un approccio *data-driven*. Più è cospicuo il numero di campioni all'interno del *training set* fornito, migliori saranno le prestazioni del modello che ne deriverà, particolarmente quando i campioni dati in ingresso alla rete hanno dimensioni elevate, come nel nostro caso (forniamo immagini di 512x512 pixel, in formato RGB - che comporta un'ulteriore moltiplicazione per 3). Se da un lato addestrare una rete convoluzionale

disponendo di sole poche decine di campioni risulta particolarmente arduo, qualche centinaio può essere sufficiente se il modello è piccolo, ben regolarizzato ed il *task* richiesto non eccessivamente arduo da portare a compimento.

5.3.1 Preelaborazione dei dati

Abbiamo bisogno di preelaborare opportunamente i dati a nostra disposizione, convertendoli in tensori in virgola mobile, prima di poterli fornire in ingresso alla rete. Dopo aver caricato il dataset sul quale intendiamo lavorare su *Google Drive*, abbiamo fatto uso di *image_dataset_from_directory*, funzione di utilità offerta da *keras* che facilita questo passaggio permettendo di configurare una pipeline di dati in grado di trasformare automaticamente i file immagine su *Drive* in *batch* di tensori preelaborati. Avendo organizzato i dati come precedentemente descritto (figura 5.4), osserviamo quanto segue:

```
1 PATH = 'classes/Polso/sys/train/'
2 image_dataset_from_directory(PATH)
```

L'ultima linea di codice eseguirà in maniera tale da listare le sottocartelle presenti in *PATH* (*high*, *normal*), assumendo che ognuna di esse contenga immagini appartenenti ad una specifica classe. Proseguirà poi indicizzando i file di ciascuna sottodirectory. Il risultato è la creazione di un oggetto *tf.data.Dataset*, che permette di elaborare i file al suo intero. Grazie all'utilizzo di *keras* ci sarà dunque possibile con facilità leggere, rimescolare, ridimensionare, suddividere in *batches* e decodificare in tensori tutti i dati caricati, a formare il nostro *dataset*. Un oggetto di questo tipo si può oltretutto passare direttamente al metodo *fit()* di un modello *keras*.

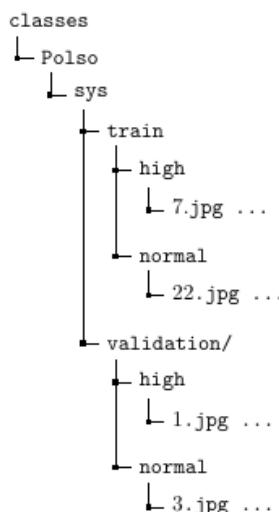


Figura 5.4

Lo scopo ultimo nella costruzione di un modello di classificazione è quello di riuscire a sottoporlo ad un addestramento tale da impartirgli la capacità di classificare campioni differenti da quelli usati in fase di addestramento. Per valutare la capacità di generalizzazione, l'intero dataset è stato separato, come già in parte accennato, in:

- *Training Set*
 - *training set* : utilizzato durante l'addestramento del modello al fine di aumentarne l'esperienza. Un *optimizer* trova la configurazione di parametri che minimizza il *training error*
 - *validation set* : è un insieme di campioni non impiegati per addestrare il modello. I parametri del modello vengono modificati tenendo conto delle performance dell'algoritmo su questo *set* di dati a ciascuna *epoch*
- *Test Set* : usato per misurare l'effettiva *performance* raggiunta dal modello (e dunque la sua capacità di generalizzazione) a fine addestramento

Disponendo di un unico campione (dei 456 iniziali) con valore di pressione sistolica arteriosa che rientrava nel range basso (*low*) di misurazione, abbiamo rimosso tale valore, riducendo il problema ad un *problema di classificazione binario* (avente per classi *normal* e *high*). Osserviamo dunque di seguito come si è proceduto alla creazione di *train_dataset* e *validation_dataset*

```
1 from tensorflow.keras.utils import image_dataset_from_directory
2
3 sys_dir = 'classes/Polso/sys/'
4 BATCH_SIZE = 32
5 IMAGE_HEIGHT = 512
6 IMAGE_WIDTH = 512
7
8 train_dataset = image_dataset_from_directory(
9     os.path.join(sys_dir, 'train'),
10    image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
11    batch_size=BATCH_SIZE
12 )
13
14 validation_dataset = image_dataset_from_directory(
15     os.path.join(sys_dir, 'validation'),
16     image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
17     batch_size=BATCH_SIZE
18 )
```

```
1 Found 364 files belonging to 2 classes.
2 Found 91 files belonging to 2 classes.
```

5.4 La Rete

5.4.1 Definire il modello

Valuteremo quattro differenti modelli, cui ci riferiremo semplicemente come:

- *Modello 0 : Modello Base*, su cui osserveremo il verificarsi di un forte fenomeno di *overfitting*. Rappresenta il modello di base su cui sono costruiti tutti i modelli che seguono. È stato realizzato a partire da una serie di prove durante le quali si è lavorato variando numero di filtri, unità per strati densamente connessi, funzioni di perdita ed ottimizzatori al fine di massimizzarne le performance.
- *Modello 1*, realizzato a partire dal precedente e attraverso il quale cerchiamo di superare i limiti posti al modello dall'*overfitting* grazie all'utilizzo di un *data_augmentation layer*.
- *Modello 2*, realizzato al medesimo scopo. Facciamo però qui - al fine di combattere l'*overfitting* - utilizzo di *dropout layers*.
- *Modello 3 : Modello Regolarizzato*, dove mettiamo insieme le tecniche di cui abbiamo fatto uso separatamente nella costruzione dei *Modelli 1 e 2*.

5.4.2 Metriche di Validazione del Modello

Ogni *classificatore binario* possiede *due* classi

- una classe *positiva*
- una classe *negativa*

Nel caso in esame lavoriamo, rispettivamente, sulle classi '*high*' e '*low*'. Illustriamo nel seguito della sezione alcune metriche (*metrics*) di validazione del modello (*model evaluation*) che utilizzeremo e che permettono di descrivere in maniera oggettiva quanto il classificatore sia efficace nel descrivere la realtà, dunque nell'identificare una determinata immagine come appartenente al *set* relativo a misurazioni che rientrano nel range basso piuttosto che in quello alto, e viceversa. Tali parametri risultano particolarmente utili dal momento che trattiamo un *imbalanced classification problem*, dove non disponiamo di un *dataset* bilanciato, particolarmente considerando *validation_dataset* (che conta 70 campioni per la classe *normal* e soli 21 per la classe *high*).

5.4.3 F1 Score

Precisione (P) La *precisione* (*precision*) rappresenta l'accuratezza con cui il sistema di apprendimento automatico prevede le classi positive. È definito come il rapporto tra i *true positive* e la somma tra *true positive* e *false positive*.

$$P = \frac{\text{true_positive}}{\text{true_positive} + \text{false_positive}} \quad (5.1)$$

Questo parametro da solo non è sufficiente, basterebbe infatti in alcuni casi avere un'unica previsione positiva corretta per avere una precisione del 100%, in altri casi sarebbe sufficiente non avere alcuna previsione falsa positiva. Non viene considerato, nel solo calcolo della *precisione*, quanto il modello stia sbagliando. Per questo la *precision* viene associata ad un'ulteriore metrica: il *recupero*

Recupero (R) Il *recupero* (*recall*) indica il rapporto di istanze positive correttamente individuate dal modello: è il numero di *true positive* diviso il numero di tutti i test che sarebbero dovuti risultare positivi (*true positive* più *false negative*).

$$R = \frac{\text{true_positive}}{\text{true_positive} + \text{false_negatives}} \quad (5.2)$$

L'*F1 score*, anche noto come *coefficiente di Dice*, rappresenta l'accuratezza di un test nell'analisi statistica nell'ambito della classificazione binaria. La misura tiene in conto *precisione* e *recupero* del test.

F1 Score L'*F1 Score* viene calcolato tramite la media armonica (il reciproco della media aritmetica dei reciproci) di *precision* e *recall* e pertanto assume valori compresi nell'intervallo [0, 1].

$$F_1 = \frac{2}{\frac{1}{R} + \frac{1}{P}} = 2 \cdot \frac{P \cdot R}{P + R} \quad (5.3)$$

La media *armonica* attribuisce un peso maggiore ai valori piccoli. Ciò fa sì che un classificatore ottenga un alto *F1 Score* solo quando *precisione* e *recupero* sono entrambi alti. Disponendo di una distribuzione irregolare nel numero di elementi appartenenti alle due classi, ci concentriamo sui risultati ottenuti guardando ai valori relativi all'*F1 score* piuttosto che a quelli relativi all'accuratezza (*accuracy*) del modello. L'*accuratezza* infatti calcola la frequenza con cui le previsioni corrispondono alle etichette: crea due variabili locali - *total* e *count* - e restituisce questa frequenza come il rapporto tra le due, non tenendo conto dei pesi relativi a ciascuna classe (dati dal numero di istanze effettivamente disponibili per ciascuna delle due, la cui differenza è importante guardando al *validation_dataset* su cui lavoriamo). Per questo ci concentreremo, nell'analisi del modello, sui valori di *F1 Score* piuttosto che sull'accuratezza.

5.4.4 Matrice di Confusione

La *matrice di confusione* fornisce un’ulteriore rappresentazione dell’accuratezza della classificazione. Ogni colonna della matrice rappresenta i valori predetti, mentre ogni riga rappresenta i valori reali. L’elemento sulla riga i -esima e sulla colonna j -esima rappresenta il numero di casi in cui il classificatore ha classificato la classe “vera” i come classe j . Attraverso questa matrice è osservabile il livello di “*confusione*” nella classificazione delle diverse classi. Sulla diagonale principale troviamo il numero di istanze correttamente classificate per ciascuna classe (*veri positivi* e *veri negativi*), su quella secondaria il numero di errori compiuti dal modello (*falsi positivi* e *falsi negativi*). Di seguito è riportato il codice utilizzato per stampare i valori delle metriche appena citate, che troveremo in più occasioni più avanti, nelle valutazioni dei vari modelli testati

```

1 import pandas as pd
2 import sklearn
3 from sklearn.metrics import classification_report, confusion_matrix
4
5 predictions = model.predict(validation_dataset).round()
6 x = confusion_matrix(Y, predictions)
7
8 matrix = pd.DataFrame(x, columns=['pred_high', 'pred_normal'], index=[

9     'HIGH', 'NORMAL'])
10
11 print('\nConfusion Matrix\n')
12 print(matrix)
13
14 print('\nClassification Report\n')
15 print(classification_report(Y, predictions))

```

5.4.5 Modello Base

Dopo aver importato *tensorflow* e *keras*

```

1 import tensorflow as tf
2 from tensorflow import keras
3 from keras import layers

```

proseguiamo con la definizione del *modello base*

```

1 model = tf.keras.Sequential([
2     tf.keras.layers.Input(shape=(128, 128, 3)),
3     tf.keras.layers.Rescaling(scale=1./255, offset=0.0),
4     tf.keras.layers.Conv2D(filters=16, kernel_size=3, activation="relu"),
5     tf.keras.layers.MaxPooling2D(pool_size=2),
6     tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation="relu"),
7     tf.keras.layers.MaxPooling2D(pool_size=2),

```

Model: "MODEL_0"		
Layer (type)	Output Shape	Param #
rescaling_5 (Rescaling)	(None, 128, 128, 3)	0
conv2d_15 (Conv2D)	(None, 126, 126, 16)	448
max_pooling2d_15 (MaxPooling2D)	(None, 63, 63, 16)	0
conv2d_16 (Conv2D)	(None, 61, 61, 32)	4640
max_pooling2d_16 (MaxPooling2D)	(None, 30, 30, 32)	0
conv2d_17 (Conv2D)	(None, 28, 28, 64)	18496
max_pooling2d_17 (MaxPooling2D)	(None, 14, 14, 64)	0
flatten_5 (Flatten)	(None, 12544)	0
dense_15 (Dense)	(None, 16)	200720
dense_16 (Dense)	(None, 16)	272
dense_17 (Dense)	(None, 1)	17

Total params: 224,593
Trainable params: 224,593
Non-trainable params: 0

Figura 5.5: Modello Base

```

8  tf.keras.layers.Conv2D(filters=64, kernel_size=3, activation="relu"),
9  tf.keras.layers.MaxPooling2D(pool_size=2),
10 tf.keras.layers.Flatten(),
11 tf.keras.layers.Dense(16, activation = "relu"),
12 tf.keras.layers.Dense(16, activation = "relu"),
13 tf.keras.layers.Dense(1, activation = 'sigmoid')
14 ])

15
16 model.compile(
17 loss='binary_crossentropy',
18 optimizer='adam',
19 metrics=['accuracy']
20 )

21
22 # Base Model
23 model._name = "MODEL_0"

```

Abbiamo iniziato a definire il modello a partire da un *Rescaling layer*, che permette di

ridurre il range iniziale dei valori che compongono ciascun dato in ingresso (ricordiamo che ogni immagine è composta da un insieme di singoli pixel che sappiamo cadere nell'intervallo $[0, 255]$) a $[0, 1]$. Una volta normalizzato il range di variazione delle caratteristiche (*features*) del dataset, costruiamo la rete convoluzionale come una pila di *layer Conv2D* (facendo uso del *rettificatore - ReLU* - come funzione di attivazione) e *MaxPooling2D* alternati. Così facendo aumenta la capacità della rete ed allo stesso tempo viene ridotta la dimensione delle *feature maps*, così che non siano eccessivamente grandi una volta raggiunto il *Flatten layer*. Iniziamo con immagini di dimensione 128x128 e, come visibile da quanto stampato a video dal metodo *model.summary()* (figura 5.5), otteniamo *feature maps* di dimensione 14x14 prima del *Flatten layer*. La profondità delle *feature maps* cresce progressivamente nella rete (da 16 a 64), mentre la loro dimensione diminuisce (da 128x128 a 14x14). Quanto appena osservato è uno schema comune nella maggior parte delle reti convoluzionali. Due *layer* densamente connessi (di 16 unità ciascuno, linee 11 – 12) precedono l'ultimo (*output layer*), che sarà composto da un singolo neurone (*Dense layer* di dimensione 1), con *funzione sigmoidea* come *funzione di attivazione* (linea 13), che codificherà la probabilità che la rete stia guardando ad una classe piuttosto che all'altra. Facciamo poi uso di *adam* come ottimizzatore (*optimizer*, linea 18) e, dal momento che il problema, come detto, è di natura binaria, usiamo la funzione di *entropia incrociata binaria (binary crossentropy)* come funzione di perdita (*loss*, linea 17).

In figura 5.6, dopo l'addestramento del modello (effettuato con chiamata a *model.fit()*), abbiamo tracciato i dati relativi a *perdita* ed *F1 score* su *train_dataset* e *validation_dataset*: sulle ascisse troviamo le *epochs*, sulle ordinate i rispettivi valori per i due sottoinsiemi di dati passati, secondo la legenda. I grafici evidenziano segni di *overfitting*. La precisione su *train_dataset* cresce linearmente nel tempo, mentre quella sul *validation_dataset* si stabilizza su un valore inferiore. La perdita (*loss*) sul *validation set* raggiunge il minimo dopo solo 5 – 10 *epochs*, dopodiché va crescendo, mentre la *training loss* continua a diminuire linearmente, avvicinandosi sempre di più allo zero. Dal momento che disponiamo di pochi campioni per l'addestramento, l'*overfitting* rappresenta il maggiore problema su cui porre la nostra attenzione. Tra le tecniche che possono aiutare a mitigare questo sconveniente fenomeno, specifiche nell'ambito della *computer vision* ed usate su larga scala durante l'elaborazione di immagini con modelli di *deep learning*, troviamo :

- Early Stopping
- Data Augmentation
- Dropout

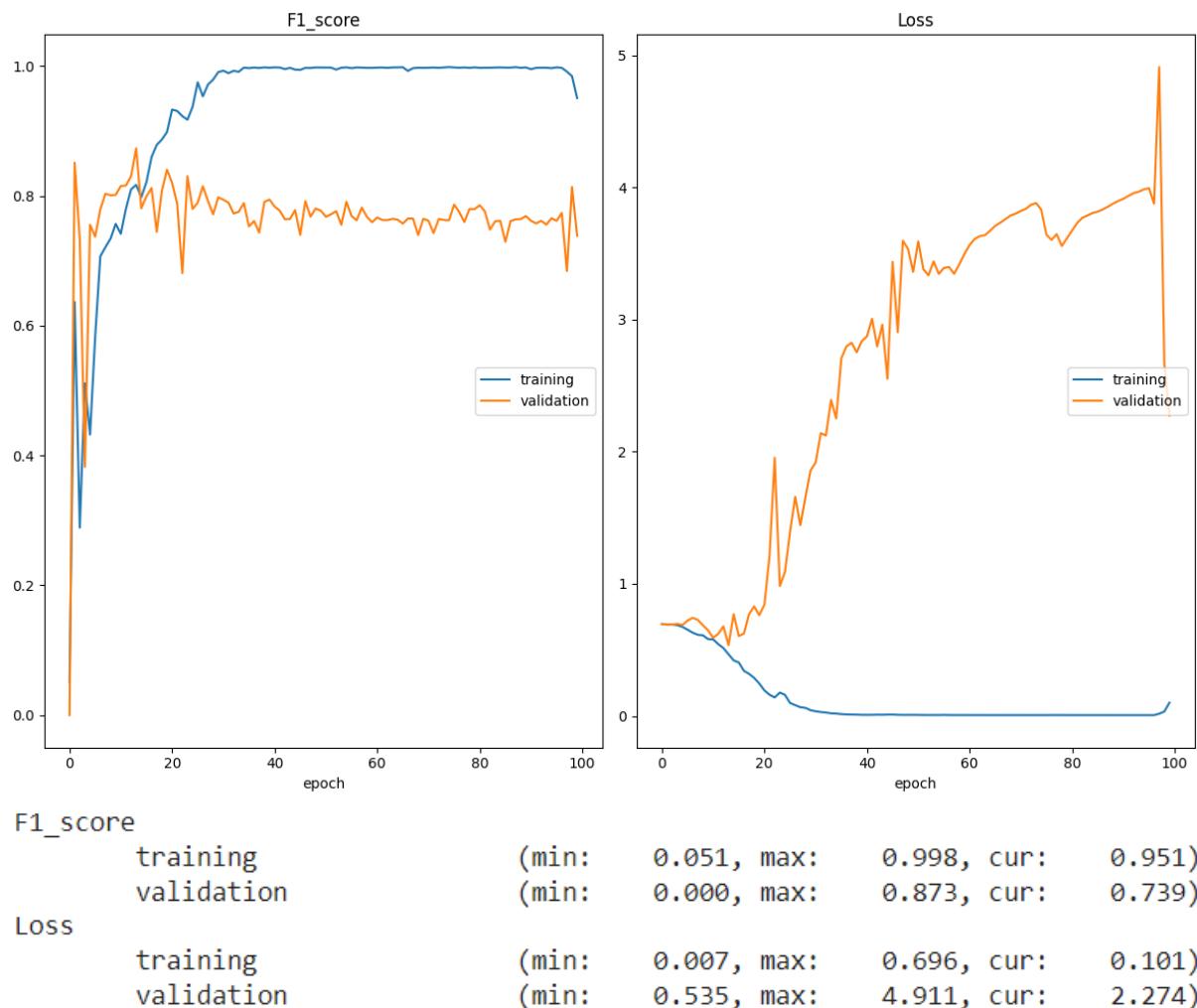


Figura 5.6: MODEL_0 : modello base

5.4.6 Modello Regolarizzato

Early Stopping

Una forma di regolarizzazione di uso frequente è rappresentata dall'*early stopping*, cui abbiamo accennato nel capitolo di introduzione alle reti neurali. Alla fine di ogni epoca monitoriamo un particolare parametro (accuratezza, perdita o f1-score) della classificazione sui dati di validazione; quando la quantità monitorata smette di migliorare per un certo numero di *epochs* consecutive (da specificare attraverso il parametro *patience*), il modello blocca l'addestramento. Questa tecnica consente di combattere l'*overfitting* regolando automaticamente il numero di *epoch*e di addestramento, cui viene settato un limite dal valore assegnato al parametro *epochs* del metodo *fit()*. L'*early stopping* (arresto anticipato) può essere facilmente implementato in *tf.keras* grazie al *callback* dedicato, ovvero un'utilità chiamata in determinati punti durante l'addestramento del modello. La definiamo come segue :

```

1 early_stop = tf.keras.callbacks.EarlyStopping(
2     start_from_epoch=30,
3     patience=10,
4     monitor='val_loss'
5 )

```

Ad ogni chiamata del metodo *fit()* faremo uso della tecnica appena illustrata inserendo *early_stop*, così come è stato definito, tra i parametri da assegnare a *callbacks*.

Data Augmentation

L'*overfitting* è causato dal nostro disporre di un numero così limitato di campioni: il modello, chiaramente, è esposto a tanti più possibili aspetti della distribuzione dei dati quanti più sono i campioni che gli vengono forniti; Averne a disposizione pochi rende difficoltoso per il modello generalizzare su nuovi dati. Il processo di *data augmentation* consiste nel generare più *training data* a partire dal *training set* di cui si dispone, "aumentando", appunto, i campioni tramite una serie di trasformazioni casuali e regolabili attraverso una serie di indici, che producono immagini dall'aspetto credibile. L'obiettivo è quello di non mostrare mai al nostro modello la stessa identica immagine durante l'addestramento al fine di esporlo a più aspetti possibile dei dati, permettendogli così di raggiungere una migliore performance di generalizzazione. In *keras* ciò può essere fatto aggiungendo una serie di *data augmentation layers* all'inizio del nostro modello. Nel definire *MODEL_1* ne includiamo uno immediatamente prima del *Rescaling layer* (guardare la definizione relativa al *Modello Base - MODEL_0*, sopra).

```

1 data_augmentation = tf.keras.Sequential([
2     layers.RandomFlip("horizontal"),

```

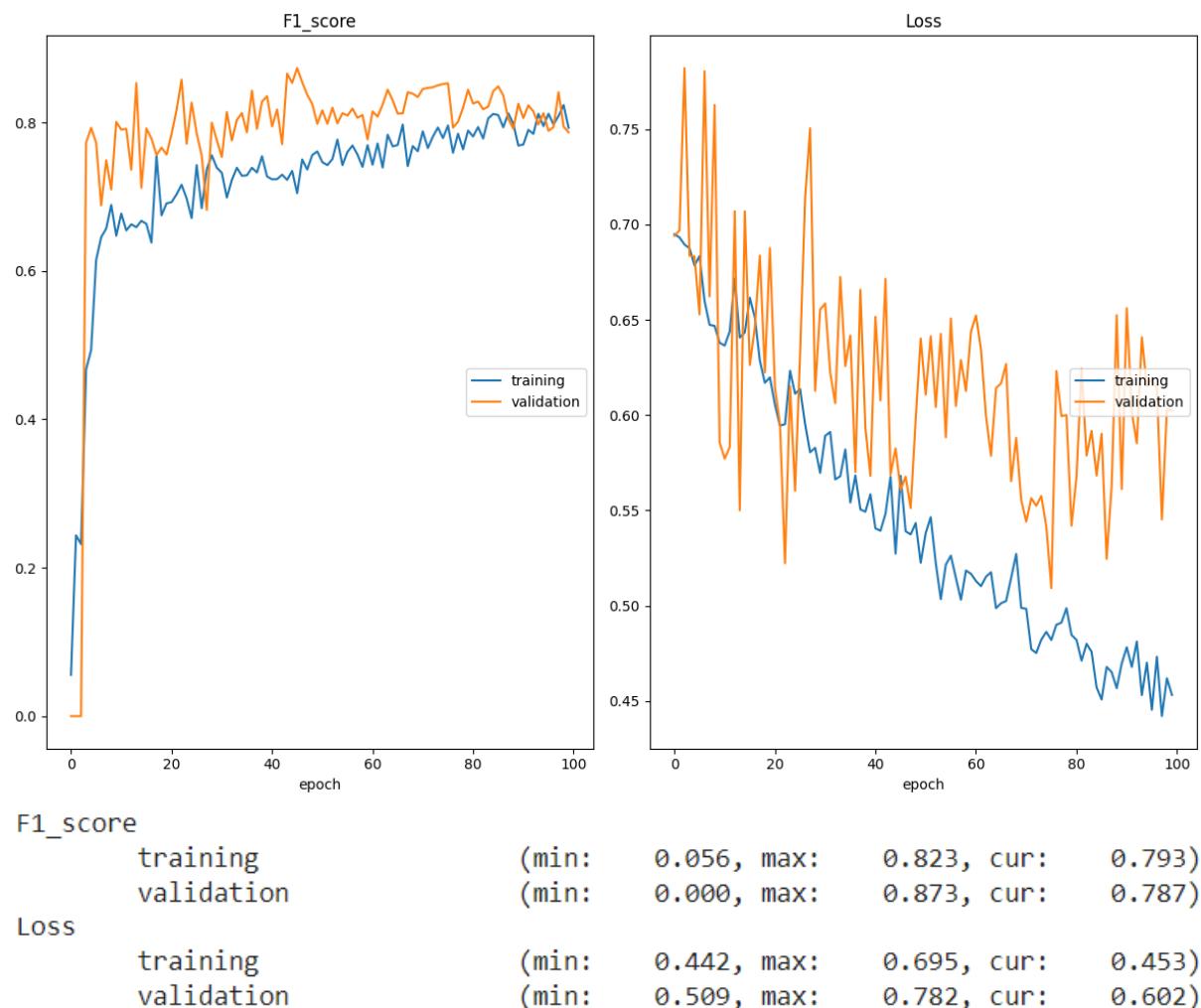


Figura 5.7: MODEL_1 : uso di data augmentation

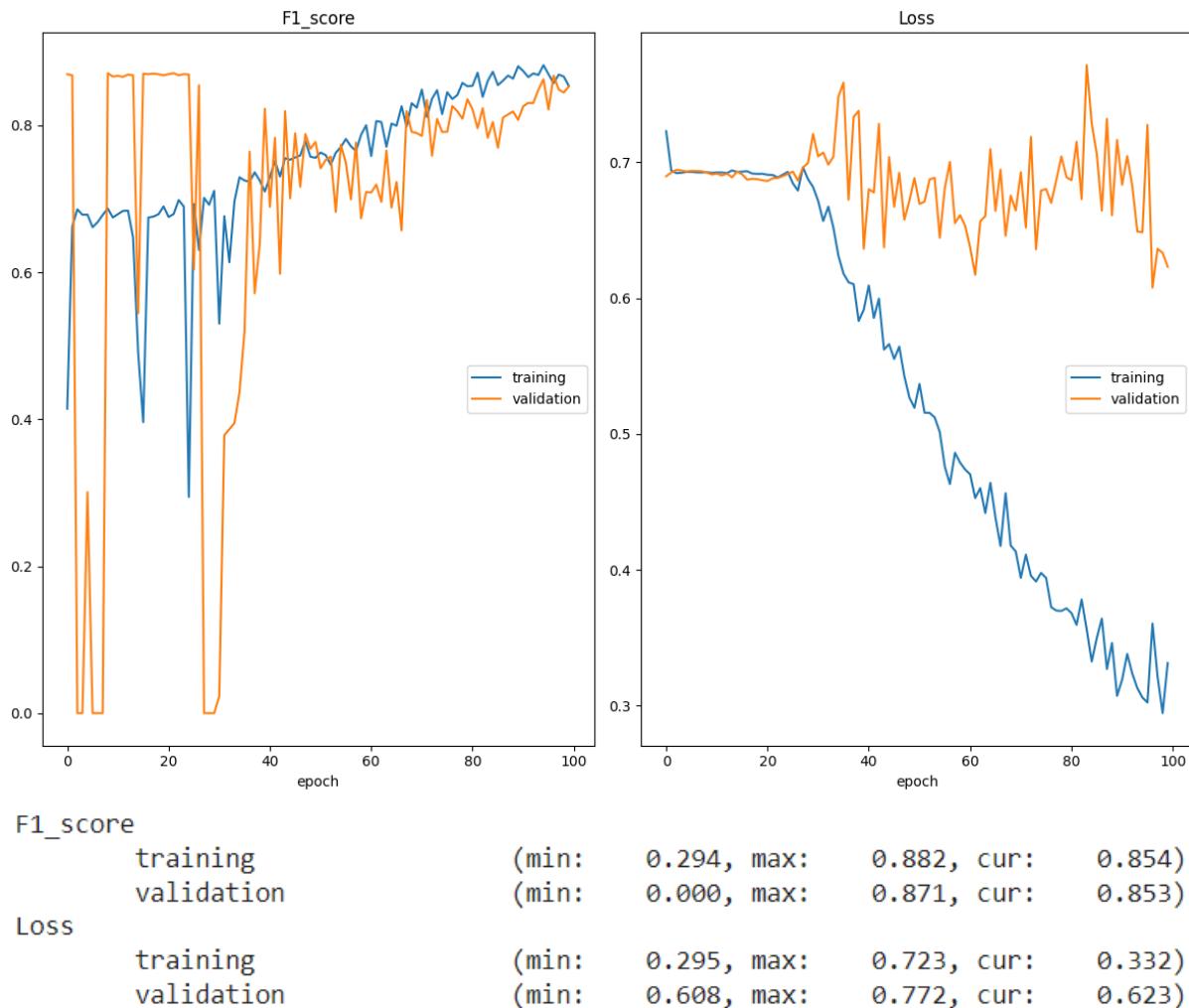


Figura 5.8: MODEL_2 : uso di dropout layers

```
3     layers.RandomRotation(0.2),
4 ])
```

Addestrando la rete con questa configurazione, non la sottoporremo mai all'analisi dello stesso identico input. Questi rimangono tuttavia fortemente collegati, provenendo comunque dal medesimo *set* di immagini originale: non possiamo produrre nuove informazioni, solo utilizzarne di già esistenti, sfruttandole a nostro favore. Il processo potrebbe dunque non essere abbastanza per eliminare l'*overfitting*. Per combatterlo ulteriormente proviamo ad aggiungere dei livelli di *Dropout* al nostro modello, prima del classificatore densamente connesso. Osserviamo in figura 5.7 i risultati derivanti dall'addestramento della rete fornita di *data augmentation layer*.

Dropout

Forniamo dunque un’ulteriore forma di regolarizzazione al nostro modello facendo uso dei cosiddetti *Dropout layers*, disponibili in *keras*. Applichiamo la tecnica del *Dropout* (brevemente esposta nella sezione 2.6.1 e qui ripresa), a seguito di ciascun *Convolutional Layer*: durante l’addestramento ciascun *dropout layer* ”spegnerà” con una certa frequenza le unità di input del rispettivo strato (troviamo il primo alla linea 6 del codice sotto, qui con un *rate* del 30%), il che aiuterà a prevenire il fenomeno di *overfitting*. Modifichiamo dunque la struttura del modello al fine di porre la nostra attenzione sul minimizzare l’errore di generalizzazione piuttosto che l’errore compiuto sul *training set*. In un caso come quello trattato, dove il *training set* è particolarmente ridotto, rischiamo che la nostra *deep neural network* vada presto incontro ad un fenomeno di *overfitting*. Per fronteggiare i problemi che sorgono in tale scenario sarebbe ottimale poter far uso di reti neurali che si basino su più modelli, configurati parallelamente in maniera differente. Seguire un approccio del genere tuttavia richiederebbe un costo computazionale non trascurabile al fine di addestrare e gestire modelli multipli. Un singolo modello può però, in quest’ottica, essere utilizzato per approssimare la situazione sopra descritta: un unico modello simula l’addestramento parallelo di diverse architetture. Si parla in questo caso di *dropout*. Un’operazione di *dropout* può risultare in ciò che equivarrebbe nel disporre di un largo numero di diverse architetture di rete, al fine di rendere il modello più robusto. In maniera casuale, secondo un certo *rate* specificato, un numero di nodi viene disattivato durante l’addestramento. Questo approccio offre una tecnica di regolarizzazione efficacie, conveniente e poco dispendiosa in termini di carico computazionale. Utilizzeremo dunque questo metodo al fine di ridurre l’*overfitting* e minimizzare l’errore di generalizzazione compiuto dal modello, che grazie a tale tecnica farà in modo che ogni *layer* sia trattato, ad ogni iterazione, come un *layer* con un diverso numero di nodi e diversa connettività al *layer* precedente. I risultati relativi al modello base con l’aggiunta di tre *dropout_layers* (con *rate* pari rispettivamente a 0.8, 0.5 e 0.5) sono illustrati in figura 5.8 Ogni aggiornamento volto ad ottimizzare il modello è attuato sulla base di una visione mai uguale a se stessa. Di fatto, disconnettendo (*dropping out*) un’unità la rimuoviamo temporaneamente dalla rete, insieme a tutte le sue connessioni in entrata e in uscita. Si aggiunge così una componente di rumore al processo di addestramento, che costringe i nodi all’interno di ciascun *layer* ad assumere con una certa probabilità più o meno responsabilità per gli input. Mettiamo ora insieme, nel codice di seguito, quanto visto in *MODEL_1* e *MODEL_2*, utilizzando insieme *data_augmentation* e *dropout_layers* per definire *MODEL_3*

```
1 model = tf.keras.Sequential([
2     tf.keras.layers.Input(shape=(128, 128, 3)),
3     data_augmentation,
4     tf.keras.layers.Rescaling(scale=1./255, offset=0.0),
```

```

5  tf.keras.layers.Conv2D(filters=16, kernel_size=3, activation="relu"),
6  tf.keras.layers.MaxPooling2D(pool_size=2),
7  tf.keras.layers.Dropout(0.8),
8  tf.keras.layers.Conv2D(filters=32, kernel_size=3, activation="relu"),
9  tf.keras.layers.MaxPooling2D(pool_size=2),
10 tf.keras.layers.Dropout(0.5),
11 tf.keras.layers.Conv2D(filters=64, kernel_size=3, activation="relu"),
12 tf.keras.layers.MaxPooling2D(pool_size=2),
13 tf.keras.layers.Dropout(0.5),
14 tf.keras.layers.Flatten(),
15 tf.keras.layers.Dense(16, activation = "relu"),
16 tf.keras.layers.Dense(16, activation = "relu"),
17 tf.keras.layers.Dense(1, activation = 'sigmoid')
18 ])
19
20 # Regularized Model
21 model._name = "model._name = "MODEL_3"

```

Compiliamo il modello come per i precedenti:

```

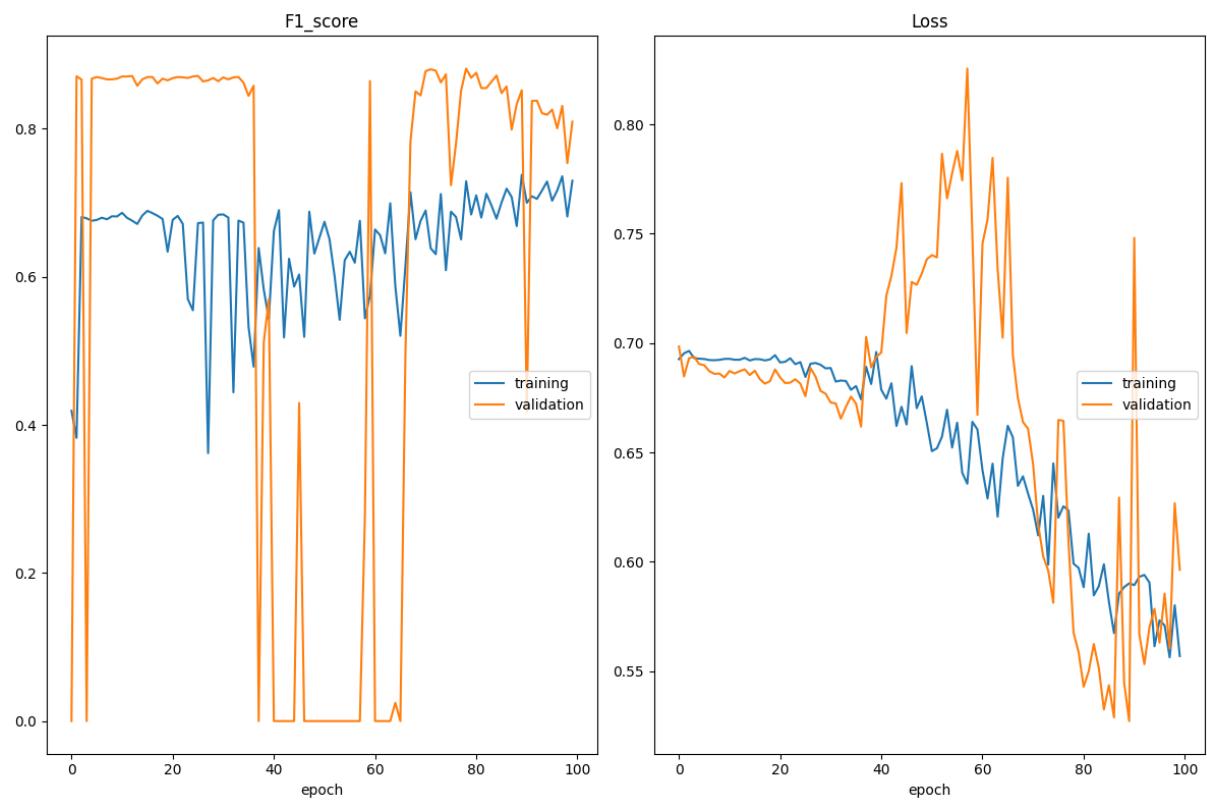
1  model.compile(
2    loss='binary_crossentropy',
3    optimizer='adam',
4    metrics=['accuracy']
5  )

```

Ed in maniera equivalente procediamo con l'addestramento del modello.

5.5 Risultati

In figura 5.9 abbiamo tracciato i dati relativi a *loss* ed *F1 score* del *modello regolarizzato* (realizzato applicando le ultime due tecniche esposte) su *train_dataset* e *validation_dataset*, facendo uso di tutte le tecniche appena discusse. Da un'analisi dei grafici risultanti dai vari processi di addestramento e dopo aver compiuto diverse valutazioni separate per ciascun modello attraverso l'utilizzo di *matrici di confusione* e *rapporti di valutazione* (grazie all'utilizzo di *sklearn.metrics*), individuiamo il *MODELLO_2* come quello che ci restituisce i migliori risultati. Se da un lato infatti il livello di *data augmentation* permette una considerevole riduzione sul valore relativo alla perdita (*loss*) sul set di validazione (basti guardare *MODELLO_1* e *MODELLO_3*, dove viene impiegato), è senz'altro il *dropout* a risultare come operazione più efficace sui nostri dati. La realizzazione della relativa rete comporta una semplice aggiunta di tre *dropout layers* rispetto al *modello base* e risulta in migliori performance sui dati di validazione e sulla minimizzazione della perdita (per entrambi i set di dati). In figura 5.10 sono stati messi dunque a confronto i risultati relativi



F1_score				
	training	(min:	0.362,	max: 0.738, cur: 0.730)
	validation	(min:	0.000,	max: 0.881, cur: 0.809)
Loss				
	training	(min:	0.556,	max: 0.696, cur: 0.557)
	validation	(min:	0.527,	max: 0.825, cur: 0.596)

Figura 5.9: MODEL_3 : uso di data augmentation layer e dropout layers

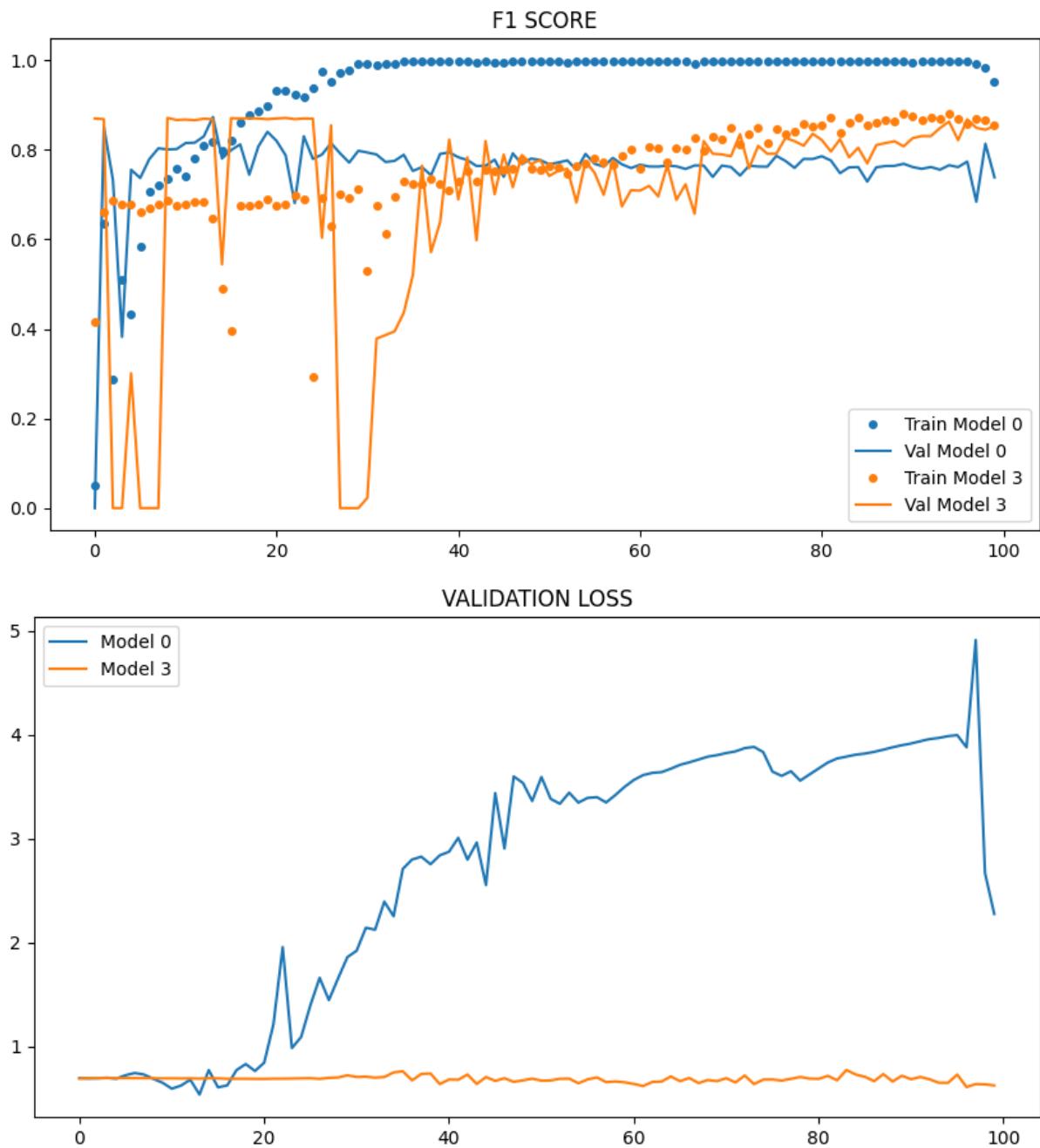


Figura 5.10

al *MODELLO_2* con quelli ricavati dall'analisi del *modello base* (figura 5.6). Riportiamo per semplicità nella seguente tabella le performance raggiunte sulle diverse metriche dai due modelli

MODEL_0						
<i>modello base</i>						
	Training			Validation		
	<i>min</i>	<i>max</i>	last	<i>min</i>	<i>max</i>	last
F1	0.051	0.998	0.951	0.000	0.873	0.739
Loss	0.007	0.696	0.101	0.535	4.911	2.274

MODEL_2						
<i>dropout</i>						
	Training			Validation		
	<i>min</i>	<i>max</i>	last	<i>min</i>	<i>max</i>	last
F1	0.294	0.882	0.854	0.000	0.871	0.853
Loss	0.295	0.723	0.332	0.608	0.772	0.623

Abbiamo evidenziato i valori migliori, considerando separatamente *training* e *validation set* su entrambi i modelli. La tecnica del *dropout*, come visibile dai valori sopra riportati (relativi alla colonna *last*, che riporta i valori riscontrati durante l'ultima *epoca* di addestramento, la 100esima per entrambi i modelli), ci permette di ottimizzare il processo di classificazione sul *validation set* ($0.853 > 0.739$, si osserva un miglioramento di circa il 10% sull'*F1 Score*), con una significativa riduzione nel valore relativo alla perdita (*loss*, $0.623 < 2.274$). Facendo uso del *MODELLO_2*, dunque, riusciamo a raggiungere un'accuracy (come misura dell'*F1 Score*), di circa l'85% sul nostro *validation set*, a fronte del circa 74% scaturito dall'addestramento sul *modello base*. Considerato il nostro fine ultimo, che corrisponde con l'impartire alla rete la capacità di associare ad un'immagine mai vista prima la corretta classe di appartenenza (il che si traduce, nel nostro caso, nell'individuare il corretto range - alto o normale - in cui cade la *pressione arteriosa sistolica* nel momento in cui viene scattata la foto che vogliamo fornire in ingresso alla rete per ottenere un risultato), come possiamo vedere, le performance raggiunte dal secondo modello sono di gran lunga superiori a quelle ottenute dal modello di partenza. Abbiamo ridotto significativamente l'*overfitting*.

Spingersi oltre tale soglia, raggiunta grazie all'applicazione dei *dropout layers*, rimane tuttavia molto irrealistico, avendo il limite di dover addestrare la nostra rete convoluzionale da zero, se non altro per il semplice fatto di disporre di troppi pochi dati su cui lavorare.

Il passo successivo verso un ulteriore miglioramento nell'accuratezza su questo problema potrebbe essere quello di sfruttare un modello pre-addestrato.

Capitolo 6

Future Applicazioni

L'intelligenza artificiale (AI) si è estesa a varie applicazioni nel settore sanitario. Si pensi alla medicina predittiva, al processo decisionale clinico, alla gestione dei dati e alla diagnostica dei pazienti. Sebbene i modelli di intelligenza artificiale abbiano raggiunto prestazioni simili a quelle umane, il loro utilizzo è ancora limitato. La causa della scarsa diffusione di questi modelli è da ricercare in quello che sostanzialmente rappresentano: una scatola nera (*black box*). Particolarmente in ambito sanitario è la mancanza di fiducia che inevitabilmente ne segue, a rappresentare la ragione principale del loro scarso utilizzo nella pratica. L'intelligenza artificiale spiegabile (*XAI, eXplainable Artificial Intelligence*) è stata introdotta come tecnica in grado di fornire fiducia nella previsione dei modelli, spiegando come una determinata previsione viene effettivamente derivata, incoraggiando così l'uso dei sistemi di intelligenza artificiale nell'assistenza sanitaria (da [10]). Il campo dell'*XAI* non è nuovo, ne troviamo traccia sin dall'origine della ricerca sull'intelligenza artificiale. A partire dal 2015 si è assistito però ad una storta di rinascimento nel suo campo di ricerca, sviluppatisi parallelamente al progresso e alla crescente prevalenza di sistemi basati sull'apprendimento automatico. Mentre l'adozione del *machine learning* continua a crescere e a raggiungere nuove fette di pubblico, modelli sempre più complessi come le *deep neural network* stanno facendo emergere nuove sfide nell'ambito della spiegabilità.

6.1 Explainable AI

Come è possibile aprire la scatola nera (*black box*) cui equivale una rete neurale, e capire che cosa sta 'pensando'? L'intelligenza artificiale, ispirata al cervello umano, promette di funzionare meglio degli algoritmi standard nel gestire le situazioni complesse del mondo reale, come può essere quella esaminata nel nostro progetto. Oltre che a ispirarsene tuttavia, le reti neurali sono anche opache quanto il cervello. Anziché memorizzare ciò che hanno appreso in un blocco ordinato di memoria digitale, distribuiscono le informazioni in

un modo molto difficile da decifrare. ([11]) Spesso vengono accettati diversi limiti, primo tra tutti quello che potremmo essere incapaci di interpretare le relazioni che guidano le decisioni dei modelli. Un ambiente regolare, come detto, è un prerequisito fondamentale per il comportamento intelligente: *azioni simili risultano in conseguenze simili*. Ogni metodo per individuare queste regolarità è necessariamente soggetto a dei limiti teorici, e di conseguenza lo è anche quello che possiamo aspettarci dagli agenti intelligenti ([01]). Quanto è possibile e giusto fidarsi dei risultati ottenuti? L'*eXplainable AI* rappresenta il grado in cui abbiamo la possibilità di comprendere modelli di intelligenza artificiale. Tale comprensione ci offre una misura della fiducia che possiamo effettivamente riporre in questi modelli. Più precisamente parliamo di un campo di ricerca che studia le tecniche di interpretabilità di modelli di *machine learning*, i cui obiettivi sono quelli di comprendere le predizioni di questi modelli e spiegarle in termini comprensibili per gli umani. Tutto ciò al fine di aumentare la fiducia dei consumatori finali dei risultati prodotti, ovvero professionisti (medici nel nostro caso) che potranno decidere, sulla base della comprensione di ciò che ha portato a un certo risultato, se servirsene o meno.

Supponiamo che il nostro modello, davanti ad un nuovo campione di *test data*, identifichi una pressione arteriosa in un particolare range. Il nostro obiettivo è quello di spiegare il perché il modello abbia scelto proprio quel range. Entriamo così nel campo dell'*eXplainable AI*. La tecnica che va sotto il nome di *integrated gradients* può evidenziare i pixel (nelle immagini sottoposte al modello come *test data*) che sono funzionali al modello nella scelta finale, dandoci un'idea di ciò che "vede" e guida le scelte della nostra rete neurale. È ottimale, in questa prospettiva, disporre di una *feature attribution mask* (più precisamente l'immagine originale cui viene sovrapposta un *IG Attribution Mask Overlay*).

Un modellista deve chiedersi come il proprio modello generalizzerà in futuro, davanti ad immagini simili a quelle su cui ha già lavorato, ma anche completamente differenti. Ciò a cui si aspira è chiaramente un modello di cui potersi fidare: vogliamo essere nella posizione di poter "difendere" i risultati offerti dal modello, spiegando cosa esattamente ha portato ad un certo output, in maniera tale da poter poi decidere se fidarci del risultato ottenuto o meno. Il prossimo passo dunque potrebbe prevedere l'integrazione nel modello di *confidence scores* (punteggi di confidenza), che sovrapposti alle *feature attribution heat maps* (mappe di calore, in grado di mettere in evidenza caratteristiche spesso perse), porterebbero ad un potenziale aumento dell'accuratezza nelle decisioni dei medici: è qui che risiede il potere del *machine learning* e dell'*eXplainable AI* nell'aumentare le performance umane. Ottenere un modello spiegabile, ovvero che spiega il *perché* ha prodotto quel particolare risultato, anziché una scatola nera che si limita a fornire un risultato, sarebbe dunque una futura prospettiva auspicabile per il progetto.

Bibliografia

- [00] Bryan Williams. "Guidelines for the management of arterial hypertension: The Task Force for the management of arterial hypertension of the European Society of Cardiology (ESC) and the European Society of Hypertension (ESH)". In: *European Heart Journal, Volume 39, Issue 33* (2018), 3021–3104.
- [01] Nello Cristianini. *LA SCORCIATOIA. Come le macchine sono diventate intelligenti senza pensare in modo umano.* il Mulino, 2023.
- [02] Chiara Donfrancesco e Simona Giampaoli reparto di Epidemiologia delle malattie cerebro e cardiovascolari Cnesps-Iss Barbara De Mei Unità di comunicazione e formazione Cnesps-Iss A cura di Serena Vannucchi Luigi Palmieri. *L'impatto della pressione arteriosa sulla salute.* <https://www.epicentro.iss.it/cardiovascolare/>.
- [03] Ltd. OMRON HEALTHCARE Co. *RS7 Intelli IT HEM-6232T-E.*
- [04] Shimmer Research Ltd. *The Shimmer3 GSR+ (Galvanic Skin Response).* <https://shimmersensing.com>.
- [05] Peter Norvig Alon Halevy e Google Fernando Pereira. "The Unreasonable Effectiveness of Data". In: *EXPERT OPINION - IEEE INTELLIGENT SYSTEMS* (2009), pp. 8–12.
- [06] © Alexander Amini e Ava Amini. *MIT 6.S191: Introduction to Deep Learning.* <http://introtodeeplearning.com/>. 2023.
- [07] Gavin Taylor Christoph Studer Tom Goldstein Hao Li Zheng Xu. "Visualizing the Loss Landscape of Neural Nets". In: (2017).
- [08] Google. *Google Developers.* <https://developers.google.com/>.
- [09] TensorFlow. *Convolutional Neural Networks - Codelab: Introduction to Convolutions.* <https://developers.google.com/codelabs/>.
- [10] Silvia Seoni Prabal Datta Barua Filippo Molinari Rajendra Acharya Hui Wen Loh Chui Ping Ooi. "Application of explainable artificial intelligence for healthcare: A systematic review of the last decade (2011–2022)". In: (2022).

- [11] Davide Castelvecchi/Nature. “Possiamo aprire la scatola nera dell’intelligenza artificiale?” In: *le Scienze, edizione italiana di Scientific American* (2016).