Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Applicazione di tecniche di computer vision a un deck per il liquid handling

Tesi di Laurea Triennale

Relatore

Prof. Zanella Marco

Laureando

Lazzarin Nicola

Matricola 2042376

Anno Accademico 2023-2024

© Lazzarin Nicola, Dicembre 2024. Tutti i diritti riservati. Tesi di Laurea Triennale: "*Applicazione di tecniche di computer vision a un deck per il liquid handling*", Università degli Studi di Padova, Dipartimento di Matematica "Tullio Levi-Civita".

Ringraziamenti

Ringrazio innanzitutto il mio relatore, il professor Zanella Marco, che mi ha seguito con grande disponibilità durante tutto il periodo di stage e durante la stesura della tesi.

Ringrazio la mia famiglia, che mi ha supportato e consigliato durante questo percorso, anche nei momenti più difficili.

Infine ringrazio anche i miei amici, sia vecchi che nuovi, per le tante risate e per l'aiuto che mi hanno dato.

Sommario

Questo documento descrive il lavoro svolto durante il periodo di stage avvenuto presso M31 S.r.l. di Padova, della durata di circa trecento ore, finalizzato all' applicazione di tecniche di computer vision nella gestione di deck con fluidi biologici di vario tipo. L'obiettivo è quello di permettere al GENEQUALITY® TWENTY-FIFTY (strumentazione fornita da AB analitica) di riconoscere varie caratteristiche delle provette inserite negli appositi deck in modo automatico e accurato.

Indice

1 Introduzione **1**

1.1 L'azienda 1

1.2 Organizzazione del testo 2

2 Apprendimento automatico **3**

2.1 Introduzione 3

2.2 Paradigmi di apprendimento automatico 3

2.3 Overfitting e underfitting 4

2.4 Metriche utilizzate 5

2.4.1 Accuracy 5

2.4.2 Loss 6

2.4.3 Altre metriche 8

3 Deep learning **10**

3.1 Struttura di una rete neurale artificiale 10

3.1.1 Funzione di attivazione 13

3.1.2 Gradient descent 17

3.1.3 Ottimizzatori 18

3.2 Funzionamento CNN 23

3.2.1 Tensori 23

3.3 Layer CNN 25

3.3.1 Layer convoluzionali 25

3.3.2 Pooling layer 28

3.3.3 Dropout layer 29

3.3.4 Batch Normalization layer 29

3.3.5 Flatten 30
3.3.6 Dense 30

4 Descrizione dello stage **31**
4.1 Introduzione al progetto 31
4.2 Analisi preventiva dei rischi 32

5 Analisi dei requisiti **34**
5.1 Tracciamento dei requisiti 34
5.2 Tabelle dei requisiti 35

6 Codifica **37**
6.1 Tecnologie e strumenti 37
6.2 Progettazione 39
6.2.1 Introduzione 39
6.2.2 GENEQUALITY® TWENTYFIFTY (GQ 2050) 39
6.2.3 Dataset 40
6.3 Codifica 44
6.3.1 Introduzione 44
6.3.2 CNN primari 44
6.3.3 CNN ausiliari 47
6.3.4 CNN cestino 48

7 Testing **50**
7.1 CNN primari 50
7.2 CNN ausiliari (provette piccole) 53
7.3 CNN ausiliari (provette grandi) 54
7.4 CNN cestino 57
7.5 Osservazioni 60

8 Conclusioni **61**

Bibliografia **i**

Elenco delle figure

3.1 Paragone tra un neurone biologico e uno artificiale 11
3.2 Esempio di rete neurale artificiale 12
3.3 Esempio di rete neurale artificiale 12
3.4 Grafico della binary step function 13
3.5 Grafico della funzione di attivazione lineare 14
3.6 Grafico della ReLU 15
3.7 Grafico della sigmoide 16
3.8 Confronto fra learning rate rispettivamente troppo grande, adeguato e troppo piccolo 17
3.9 Esempi di funzioni con minimi uguali ma diversi gradienti 19
3.10 Confronto fra le derivate delle funzioni 19
3.11 Esempio di un immagine 24
3.12 Esempio di una matrice 25
3.13 Esempio effetto di bordo 27
3.14 Esempio utilizzo *padding* 27
3.15 Esempio utilizzo stride 28

6.1 Loghi delle varie tecnologie usate 38
6.2 Immagine di calibrazione per il riconoscimento delle provette 40
6.3 A sinistra le provette senza riflesso, a destra provette dello stesso tipo con riflesso 41
6.4 A sinistra tappi in posizioni anomale, a destra provette parzialmente coperte da altre provette 42

6.5 Esempi di diverse provette appartenenti alla stessa classe (tappi), in basso a destra un tappo del *rack* ausiliario 42
6.6 Esempio di un file csv per il *rack* primario, le colonne notlabeled non vengono considerate al momento del ritaglio e indicano dei *rack* mancanti 43
6.7 Esempio della struttura di uno dei modelli usati 44

7.1 Grafico dell'*accuracy* del modello al passare delle *epoch* 50
7.2 Grafico della *loss* del modello al passare delle *epoch* 51
7.3 *Confusion matrix* della CNN 52
7.4 Esempi di errori commessi dalla CNN nella classificazione 52
7.5 Esempi delle diverse provette 53
7.6 Grafico della *accuracy* del modello al passare delle *epoch* 53
7.7 Grafico della *loss* del modello al passare delle *epoch* 54
7.8 Grafico della *loss* del modello al passare delle *epoch* 55
7.9 Grafico della *loss* del modello al passare delle *epoch* 55
7.10 *Confusion matrix* della CNN 56
7.11 Esempi di errori commessi dalla CNN nella classificazione 56
7.12 Grafico della *accuracy* del modello al passare delle *epoch* 57
7.13 Grafico della *loss* del modello al passare delle *epoch* 58
7.14 *Confusion matrix* della CNN 59
7.15 Esempi di errori commessi dalla CNN nella classificazione 59

Elenco delle tabelle

5.1 Tabella del tracciamento dei requisiti funzionali. 35
5.2 Tabella del tracciamento dei requisiti qualitativi. 36
5.3 Tabella del tracciamento dei requisiti di vincolo. 36

Capitolo 1

Introduzione

1.1 L'azienda

M31 nasce nel 2007 come centro di incubazione tecnologica, con una forte base ingegneristica che nel corso degli anni si è espansa a vari ambiti creando anche degli *spinoff*, fino a diventare un partner tecnologico di SMACT – uno degli 8 Centri di Competenza industria 4.0 nati in Italia su impulso del Ministero dello Sviluppo Economico. Utilizzando gli output della ricerca scientifica, la propria rete di partner e gli ultimi avanzamenti tecnologici del mercato, M31 è in grado di sostenere le imprese nelle generazione di nuove idee di prodotto e di processo, nell'esplorazione di alternative tecnologiche e di design, nel miglioramento di prodotti e processi esistenti, nell'avvio di nuovi e rivoluzionari modelli di business soprattutto nei mercati del *Biomed*, dell'*Agritech*, delle *Smart Cities* e *Smart Energy*, dell'*Industrial Automation*. M31 è specializzata in soluzioni IoT, Cloud, Intelligenza Artificiale ed Automazione avanzata (sistemi ciberfisici, *digital twin*, simulazione ecc.) applicati all'industria e alla creazione di valore nei mercati di riferimento. Ad oggi M31 è composta da 40 professionisti che spaziano tra competenze come la progettazione *hardware, software* e *firmware* al design e dalla gestione manageriale e finanziaria d'impresa all'esplorazione e analisi di nuovi modelli di business. La sua sede è vicina alle sedi di matematica fisica e ingegneria dell'Univeristà di Padova con cui ha stretti contatti.

1.2 Organizzazione del testo

Il secondo capitolo introduce i concetti chiave dell'apprendimento automatico e discute le principali metriche utilizzate per valutare i modelli.

Il terzo capitolo approfondisce il tema del deep learning e le CNN in tutte le loro componenti

Il quarto capitolo descrive gli obiettivi dello stage e contiene l'analisi dei rischi.

Il quinto capitolo consiste nell'analisi dei requisiti.

Il sesto capitolo tratta di come sono stati creati i dataset e dell'implementazione del codice delle reti neurali.

Nel settimo capitolo si descrivono i risultati ottenuti nel testing delle varie CNN.

Nell'ottavo capitolo vengono tratte le conclusioni e proposti alcuni miglioramenti.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Apprendimento automatico

2.1 Introduzione

Tom M. Mitchell, uno dei fondatori e membro del *Machine Learning Department* della *Carnegie Mellon University*, ha fornito la definizione più accreditata di apprendimento automatico, ovvero: "Si dice che un programma apprende dall'esperienza E con riferimento a alcune classi di compiti T e con misurazione della performance P, se le sue performance nel compito T, come misurato da P, migliorano con l'esperienza E."[1]. In poche parole l'obiettivo di questa branca dell'intelligenza artificiale è creare algoritmi in grado di identificare pattern all'interno dei dati con performance sempre migliori senza bisogno di interventi dall'esterno. Questo concetto è stato applicato in diversi modi nel corso del XX secolo. Un esempio ne sono le reti neurali artificiali come quelle sviluppate e utilizzate nello svolgimento di questo stage.

2.2 Paradigmi di apprendimento automatico

L'apprendimento automatico viene diviso in 3 categorie principali in base a come si approccia ai problemi. Queste tre categorie sono dette paradigmi e sono: apprendimento supervisionato, apprendimento non supervisionato e apprendimento per rinforzo. L'apprendimento supervisionato, ovvero la tecnica usata in questo sta-

[1] Jaime Carbonell. *Machine learning.* Tom M. Mitchell, Jaime Carbonell, Ryszard Michalski, 1997.

ge, prevede che, durante l'allenamento della rete neurale, i dati forniti in input (che compongono quindi il training set) siano già opportunamente catalogati e divisi in categorie creando n coppie $(\vec{X}, y) \in R^n \times L$ con L insieme di etichette, $\vec{x}$ sarà il dato (nel caso in esame le immagini) e y la classificazione dello stesso. Durante il processo di *training* il modello apprenderà le similarità tra elementi della stessa classe e quindi imparerà a mappare nuovi elementi non visti durante la fase di *training* (il *validation set* e il *test set*) con le rispettive categorie. L'apprendimento non supervisionato invece non richiede una preparazione a monte dei dati: infatti è sufficiente fornire i dati non classificati (quindi solo le $\vec{x}$ delle coppie viste prima) e l'algoritmo, attraverso il cosiddetto *clustering*, estrapolerà le *feature* dai dati e le raggrupperà nelle classi. Infine il *reinforcment learning* si basa sull'interazione fra algoritmo e ambiente: ad ogni iterazione esso riceve un input i e lo stato s dell'ambiente, in base a ciò seleziona un'azione a la quale può avere tre diversi esiti a seconda di come influenza l'ambiente (e quindi in base a come vine influenzata la sua funzione obiettivo): se l'azione è positiva sarà premiata, se negativa sarà punita e se neutra non riceverà nessun feedback.

2.3 Overfitting e underfitting

Overfitting e *underfitting* sono le due problematiche più comuni e importanti nel campo dell'apprendimento automatico ma per poterle comprendere è necessario introdurre due concetti: bias e varianza. Il bias è un errore sistematico dell'algoritmo dovuto ad assunzioni durante il processo di training che portano a discriminare alcuni risultati rispetto ad altri portando a rappresentazioni dei dati non fedeli alla realtà. La varianza invece è la suscettibilità dell'algoritmo a fluttuazioni, anche minime, sui dati. L'*overfitting*, caratterizzato da un'alta varianza e un basso bias, si concretizza con l'incapacità del modello di generalizzare ciò che ha appreso per poterlo riutilizzare su dati nuovi. Le cause di ciò sono varie: il modello può essere troppo complesso, il dataset fornito può essere troppo piccolo (causando la scelta arbitraria delle feature imparate dall'algoritmo, il quale può focalizzarsi, ad esempio, sul rumore dei dati forniti, senza imparare le feature realmente importanti). L'*underfitting*, problema opposto all'*overfitting* distinto da un alto bias e una bassa

varianza, è l'incapacità di un modello di rappresentare adeguatamente i dati forniti: questo può essere causato dalla troppa semplicità del modello o da limitazioni sulle risorse computazionali.

2.4 Metriche utilizzate

2.4.1 Accuracy

Al fine di valutare quanto un modello sia efficace nel riconoscimento degli input è opportuno utilizzare alcune metriche, una delle più utilizzate è l'*accuracy*. Per comprendere il significato di questa metrica si deve fare riferimento ad una matrice di confusione, ovvero una rappresentazione di come vengono classificati gli elementi, visualizzando sulle righe i valori reali e sulle colonne i valori previsti. In particolare in una matrice 2×2 si ottengono:

***True positive* (TP)**: il numero di elementi classificati correttamente come positivi

***True negative* (TN)**: il numero di elementi classificati correttamente come negativi

***False positive* (FP)**: il numero di elementi classificati erroneamente come positivi

***False negative* (FN)**: il numero di elementi classificati erroneamente come negativi

	Previsione positiva	Previsione negativa
Dato positivo	TP	FN
Dato negativo	FP	TN

Detto questo l'*accuracy* è definibile come:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

In poche parole l'*accuracy* è il numero di previsioni corrette sul totale di tutte le previsioni. Il vantaggio principale di questa metrica è che è facilmente comprensibile

tuttavia può risultare completamente inutile nel caso in cui il *dataset* sia sbilanciato: se ad esempio abbiamo un *dataset* composto per il 90% da una classe e per il restante 10% da una seconda classe l'algoritmo potrebbe classificare banalmente ogni elemento come appartenente alla prima classe, classificando erroneamente ogni elemento della seconda ma mantenendo un accuratezza del 90%. È quindi fondamentale, se si vuole considerare questa metrica, mantenere un *dataset* bilanciato: dato che nel caso in esame il *dataset* era completamente sotto il mio controllo è stato possibile utilizzare questa metrica.

2.4.2 Loss

Altra metrica utilizzata è la *loss*: essa rappresenta la distanza fra il risultato previsto e quello reale e viene calcolata tramite la *loss function*. Le funzioni di *loss* sono separabili in 2 categorie a seconda dell' ambito di utilizzo: infatti le funzioni di *loss* per la regressione (tecnica utilizzata per predire valori continui) e per la classificazione (tecnica utilizzata per suddividere i dati in input in diverse classi a seconda delle loro caratteristiche) sono ben distinte. Un esempio della prima tipologia può essere la *quadratic loss*: più semplice dal punto di vista matematico in quanto un errore viene pesato nello stesso modo indipendentemente dalla sua posizione rispetto all'obiettivo. La sua formula è:

$$\lambda(x) = C(t - x)^2$$

dove C è una costante, generalmente posta a 1, t è il valore reale e x è il valore predetto.
Altro esempio è la *mean absolute error* (MAE) la quale si esprime matematicamente con la formula:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i|$$

dove y_i è il valore predetto, x_i è il valore reale e n il numero di elementi considerati.

Per quanto riguarda invece le funzioni di classificazione possiamo individuare: la *hinge loss*, utilizzata principalmente nelle SVM (*Support Vector Machines*) e

descritta dalla seguente formula:

$$l(y) = max(0, 1 - t \times y)$$

dove y è la predizione e t un valore pari a +1 oppure -1. Infine l'ultima categoria (quella della *cross entropy loss*) si distingue ulteriormente in *binary Corss-Entropy loss* e *categorical Cross-Entropy loss* a seconda del numero di classi da riconoscere nel modello. La *Cross-Entropy loss* calcola la differenza tra al probabilità che un certo elemento fornito in input appartenga ad una certa classe e la sua reale appartenenza alla suddetta classe. Nel caso della *binary Cross-Entropy loss* questa probabilità è compresa tra 0 e 1 in quanto le classi tra cui scegliere sono solo 2 ed è rappresentata dalla formula:

$$Loss = -\frac{1}{N}\sum_{i=1}^{N}[y_{true}log(y_{pred}) + (1 - y_{true})log(1 - y_{pred})]$$

dove N è il numero di elementi y_{true} la classificazione reale del singolo esempio e y_{pred} la probabilità che un esempio appartenga ad una determinata classe. In particolare possiamo notare come $y_{true}log(y_{pred})$ calcola la *loss* della predizione per la classe positiva: tanto più il valore della predizione sarà vicino a 1 tanto più la loss sarà bassa e viceversa. Analogamente $(1 - y_{true})log(1 - y_{pred})$ calcola la *loss* relativa alla classe negativa.

Nel caso della categorical Cross-Entropy loss la formula invece diventa questa:

$$Loss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C}(y_{i,j} \times log(p_{i,j}))$$

dove N è il numero di istanze presenti nel dataset, C è il numero delle classi, $y_{i,j}$ rappresenta le etichette corrette di ogni elemento e $p_{i,j}$ è la probabilità che un elemento i appartenga alla classe j. Uno dei vantaggi di questa funzione di classificazione è che, vedendo le diverse probabilità di classificazione nelle diverse classi, si può verificare la confidenza con cui un certo elemento viene classificato e quindi ottenere una ulteriore metrica per determinare la qualità della rete neurale.

2.4.3 Altre metriche

Esistono inoltre altre metriche, le quali non sono state utilizzate per vari motivi:

Precision[2]: è una misura di quante predizioni con esito positivo sono state effettuate correttamente: più è alto il valore maggiore è il numero di *true positive*.

$$Precision = \frac{TP}{TP + FP}$$

Recall: questo valore identifica quante istanze positive sono identificate correttamente dal modello, se questo valore è alto significa che vengono identificate correttamente la maggior parte delle istanze positive.

$$Recall = \frac{TP}{TP + FN}$$

F1 score: data la presenza di un *tradeoff*[3] tra *precision* e *recall* dovuto al fatto che la *precison* da maggior importanza alla riduzione dei falsi positivi mentre la *recall* predilige la riduzione dei falsi negativi ha portato alla creazione della *F1-score*, metrica che combina le due precedenti bilanciandone vantaggi e svantaggi calcolando la media armonica tra le due metriche. La media armonica, definita come il reciproco della media dei reciproci, permette di ottenere un valore alto di F1-score solo quando entrambi i valori (*precision* e *recall*) sono alti, ottenendo così un bilanciamento.

$$F1 - score = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

Queste tre metriche non sono state utilizzate in quanto dare un'importanza maggiore ad una classe rispetto a tutte le altre sarebbe stato solo marginalmente sensato e utile (è vero che sbagliare a identificare un tappo è più dannoso

[2] *F1-score, precision, recall.* URL: `https://medium.com/@nirajan.acharya777/understanding-precision-recall-f1-score-and-support-in-machine-learning-evaluation-7ec935e8512e`.

[3] *Precision-recall tradeoff.* URL: `https://medium.com/analytics-vidhya/precision-recall-tradeoff-79e892d43134`.

che sbagliare a identificare la dimensione di una provetta in quanto potrebbe portare al danneggiamento della macchina tuttavia sbagliare troppe provette di una qualsiasi classe rende praticamente inutile il modello).

Balanced accuracy[4]: variante dell'*accuracy* utilizzata nel caso in cui le classi non siano bilanciate (ovvero una classe appare un numero di volte maggiore nel dataset). Essa è rappresentata dalla seguente formula:

$$BA = \frac{Recall + Specificity}{2}$$

dove *specificity* è pari a:

$$Specificity = \frac{TN}{TN + FP}$$

Nel caso in esame non è stato necessario considerarla perché è stata posta particolare attenzione a mantenere le classi bilanciate

[4] *Balanced accuracy.* URL: `https://www.statology.org/balanced-accuracy/`.

Capitolo 3

Deep learning

Il *deep learning* è una branca dell'apprendimento automatico caratterizzata dall'utilizzo di reti neurali con molti *hidden layer* addestrate su grandi quantità di dati, generalmente immagini.

3.1 Struttura di una rete neurale artificiale

Le reti neurali convoluzionali (CNN) sono delle reti neurali artificiali, ovvero un modello computazionale utilizzato nel campo dell'apprendimento automatico, la cui struttura è ispirata a strutture biologiche come, ad esempio, la corteccia visiva degli umani. Esse vengono impiegate principalmente nei sistemi di raccomandazione, nell'elaborazione del linguaggio naturale, in bioinformatica e nel riconoscimento di immagini e video. Proprio grazie alla loro applicabilità in quest'ultimo campo sono state scelte per lo svolgimento di questo stage. La struttura delle reti neurali, di cui le CNN sono un sottoinsieme, è composta da *layer* a loro volta composti da neuroni, i quali, come detto precedentemente, prendono spunto nella loro struttura dalle controparti biologiche come si può vedere nella Fig. 3.1. Infatti possiamo individuare per ogni neurone un input (il corrispettivo dei dendriti), un output (il corrispettivo dell'assone) e infine il "corpo" del neurone, il cui scopo è elaborare gli input tramite la cosiddetta funzione di attivazione.

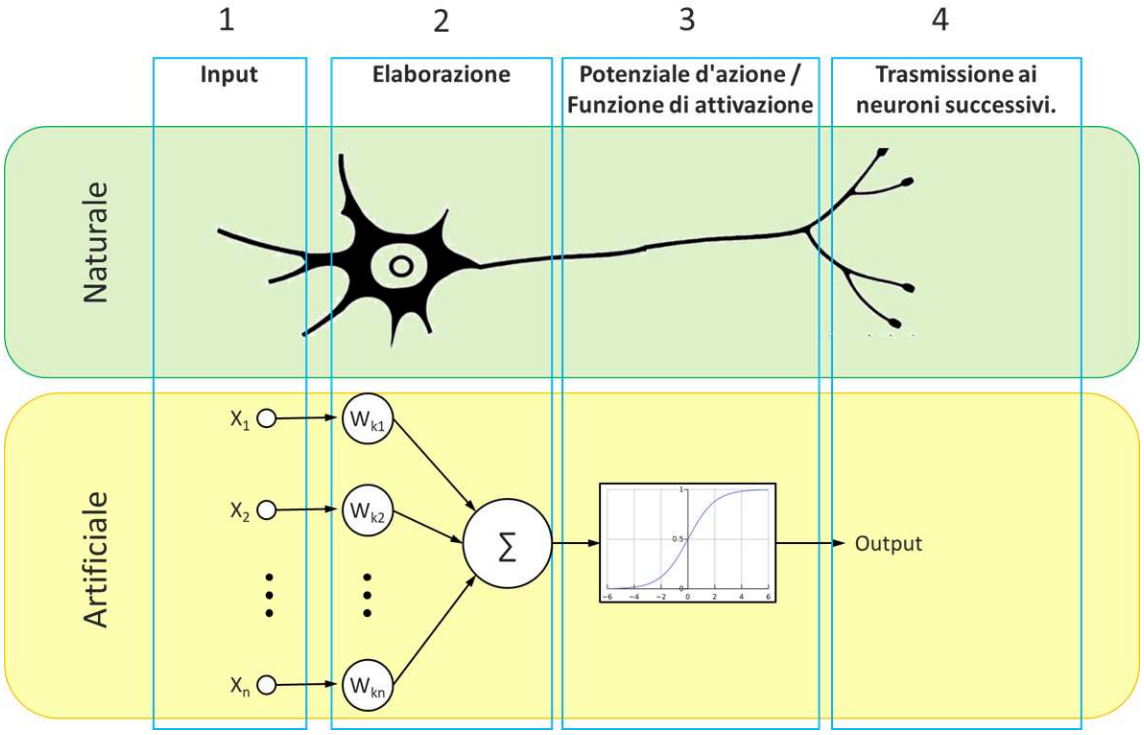

Figura 3.1: Paragone tra un neurone biologico e uno artificiale

Come detto questi neuroni sono disposti in *layer*, i quali possono essere suddivisi in tre tipologie: i *layer* di input e output i quali sono presenti una sola volta, rispettivamente all'inizio e alla fine della rete e i cosiddetti *hidden layer* il cui compito è elaborare i dati ricevuti in input, i quali verranno elaborati diversamente in base al numero di neuroni e di connessioni fra essi. Queste differenze strutturali portano alla formazione di diversi tipi di reti neurali, ad esempio se le connessioni vanno solo in una direzione (dall'input verso l'output), come nel caso delle reti sviluppate durante questo stage, allora la rete è detta *feed-forward* (di cui si può vedere un esempio nella Fig. 3.2), in caso contrario (ovvero con la presenza di cicli) avremo le cosiddette reti neurali ricorrenti. Ognuno di questi *layer* effettuerà delle trasformazioni sugli input ricevuti salvando le relative informazioni sotto forma di dati numerici detti pesi (anche chiamati parametri del *layer*) i quali si influenzano a vicenda. Una volta modificati i pesi il modello tenterà di effettuare una predizione, e quindi di classificare i dati forniti. La predizione verrà quindi confrontata con le classi reali dei dati calcolando il valore della *loss* tramite la relativa funzione. Infine questo valore verrà riutilizzato dall'ottimizzatore per influenzare i pesi tramite il processo di *backpropagation* (come si vede nella Fig. 3.3). È da notare inoltre come questo

processo sia utilizzato per allenare la CNN senza però alterarne la struttura che rimane quindi quella di una rete neurale *feed-forward*.

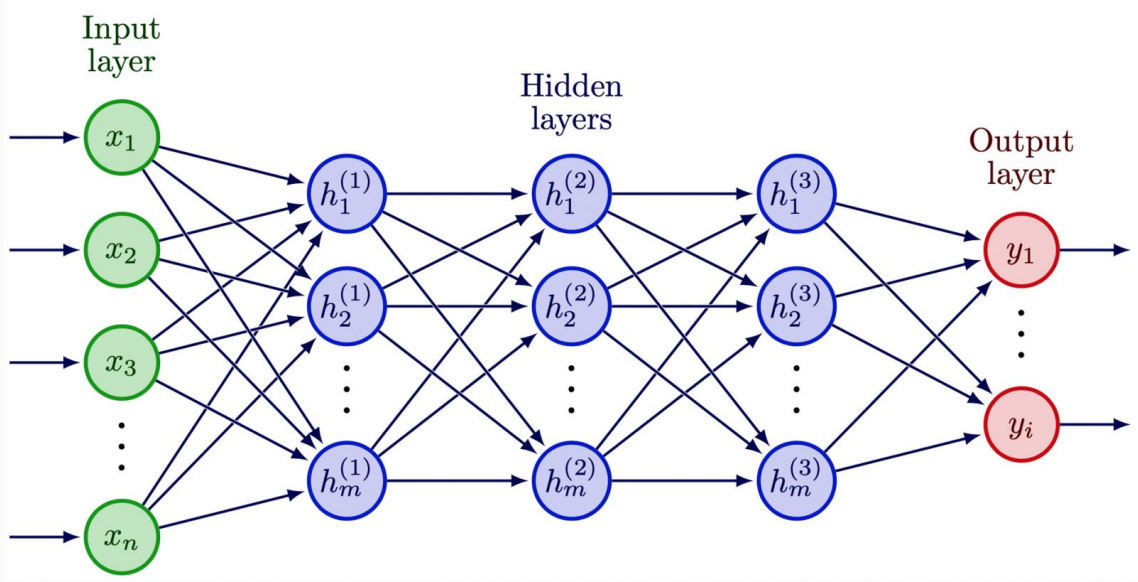

Figura 3.2: Esempio di rete neurale artificiale

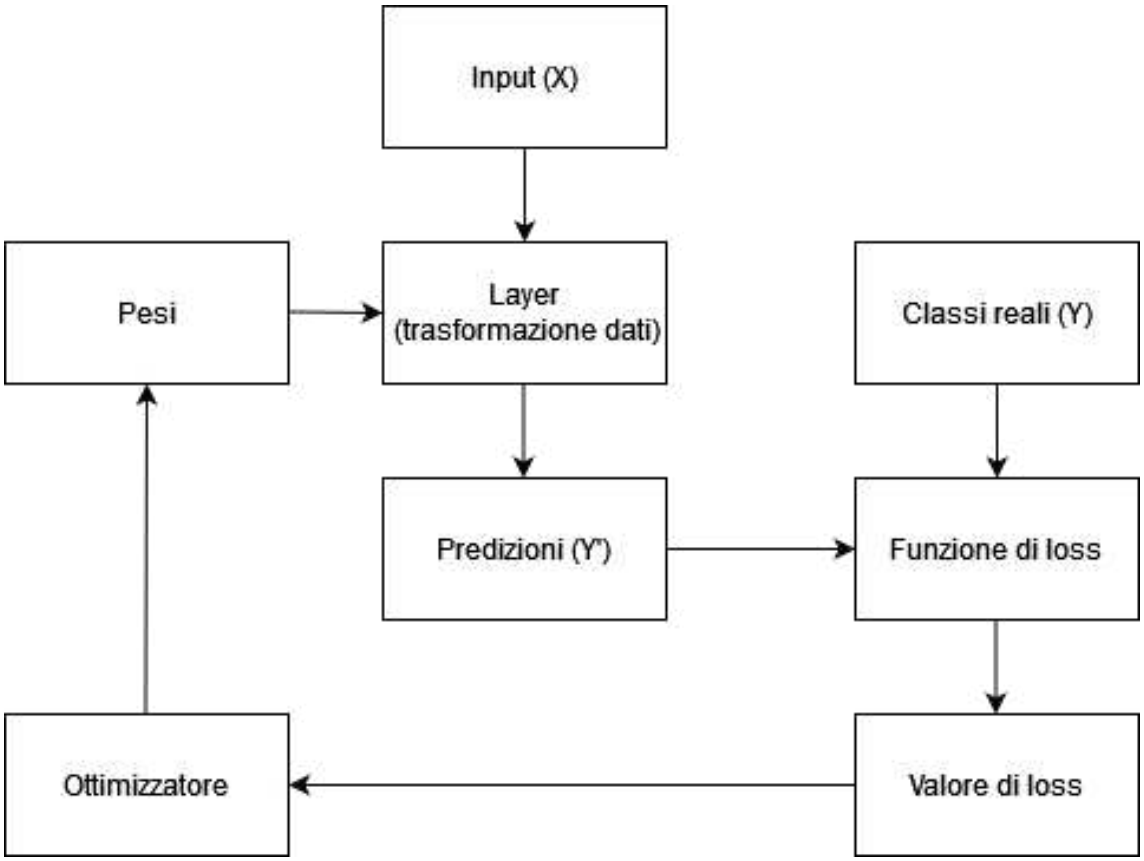

Figura 3.3: Esempio di rete neurale artificiale

3.1.1 Funzione di attivazione

La funzione di attivazione si occupa di determinare se un certo neurone debba rimanere attivo o meno basandosi sul suo input: questo permette di introdurre una maggiore complessità al modello facendo in modo che i neuroni possano comportarsi in modo diverso. Esistono 3 diverse tipologie di funzioni di attivazione: la prima, più semplice, è detta *binary step function*: essa è caratterizzata da un valore soglia che, se superato, permette di emettere un certo output come si vede in Fig. 3.4.

$$f(x) = \begin{cases} 0, & \text{se } x < 0 \\ 1, & \text{se } x >= 0 \end{cases}$$

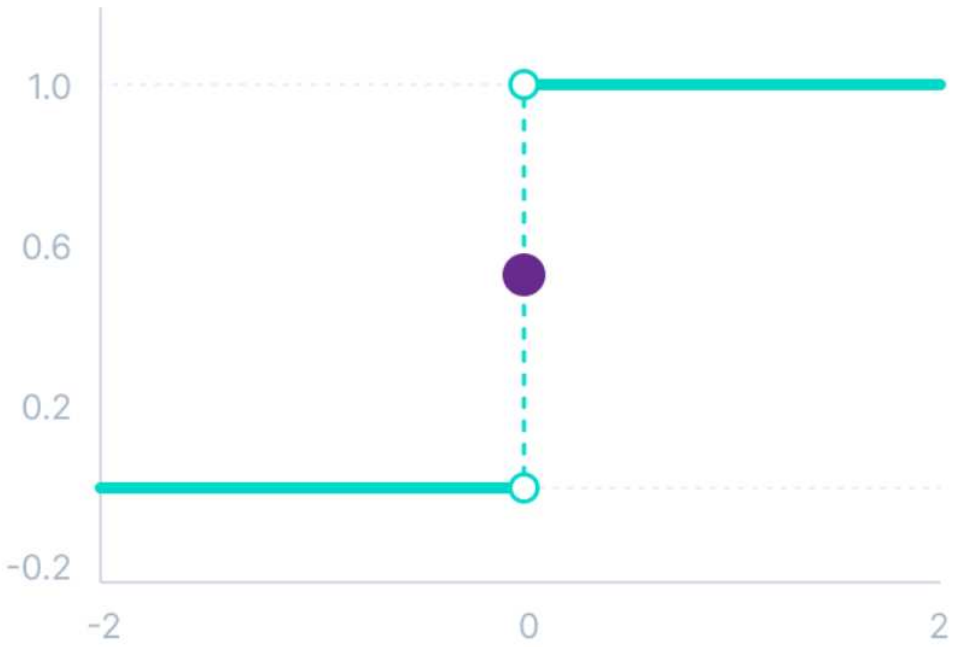

Figura 3.4: Grafico della binary step function

Questa funzione presenta due problemi principali: il primo è che può essere usata solo per operazioni di classificazione binaria in quanto è in grado di fornire solo due output, e che il secondo è il suo gradiente vettore che ha come componenti le derivate parziali della funzione) pari a zero che impossibilita l'uso di *backpropagation*.
La seconda tipologia di funzione di attivazione, il cui grafico si vede in Fig. 3.5, è quella lineare, esprimibile come:

$$f(x) = x$$

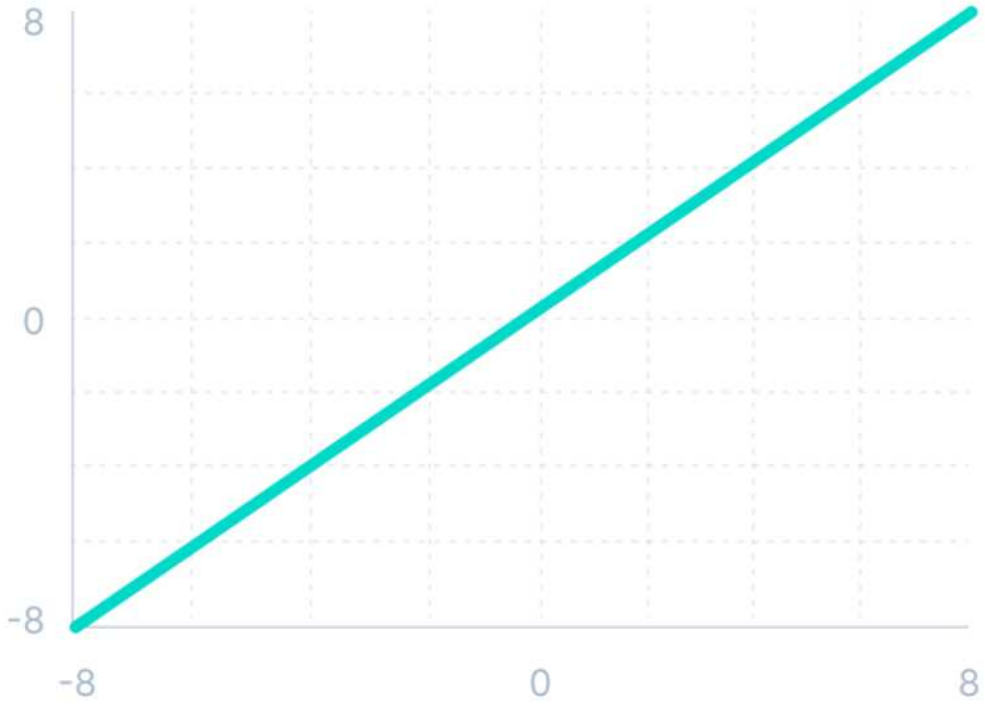

Figura 3.5: Grafico della funzione di attivazione lineare

essa non permette di utilizzare *backpropagation* e in più farà in modo che l'ultimo layer sia una funzione lineare del primo, facendo collassare tutti i layer in uno solo. Infine l'ultima categoria è rappresentata dalle funzioni non lineari le quali, ben più potenti e complesse delle funzioni lineari, permettono di utilizzare *backpropagation* e di utilizzare in modo efficace più layer. Alcuni esempi di funzioni di attivazione non lineari sono: la sigmoide (o funzione logistica), la tanh (tangente iperbolica) e la ReLU (*Rectified Linear Unit*) la quale, nonostante sembri una funzione lineare permette la *backporpagation* rendendola anche più efficiente dal punto di vista computazionale, disattivando i neuroni che ottengono un prodotto minore di zero dalla combinazione lineare degli input. Questo suo punto di forza è anche causa di una delle problematiche maggiori di questa funzione ovvero il *dying ReLU problem*[1]: questo consiste nella disattivazione consistente di vari neuroni, fattore che porta la

[1] *Dying relu.* URL: `https://www.geeksforgeeks.org/what-is-the-dying-relu-problem-in-neural-networks/`.

rete neurale a perdere la capacità di apprendere nuove informazioni a partire dai dati in quanto il gradiente per la *backpropagation* è pari a zero quando i neuroni sono disattivati. Questo può essere dovuto a una distribuzione iniziale dei pesi errata. Una delle tecniche utilizzate per risolvere questa problematica è la cosiddetta *leaky ReLu*[2], la quale implementa una leggera pendenza anche per i valori minori di 0. Tuttavia nel contesto di questo stage non è stato necessario adottarla per ottenere buone metriche. La ReLU è rappresentabile matematicamente come si vede in Fig. 3.6:

$$f(x) = max(0, x)$$

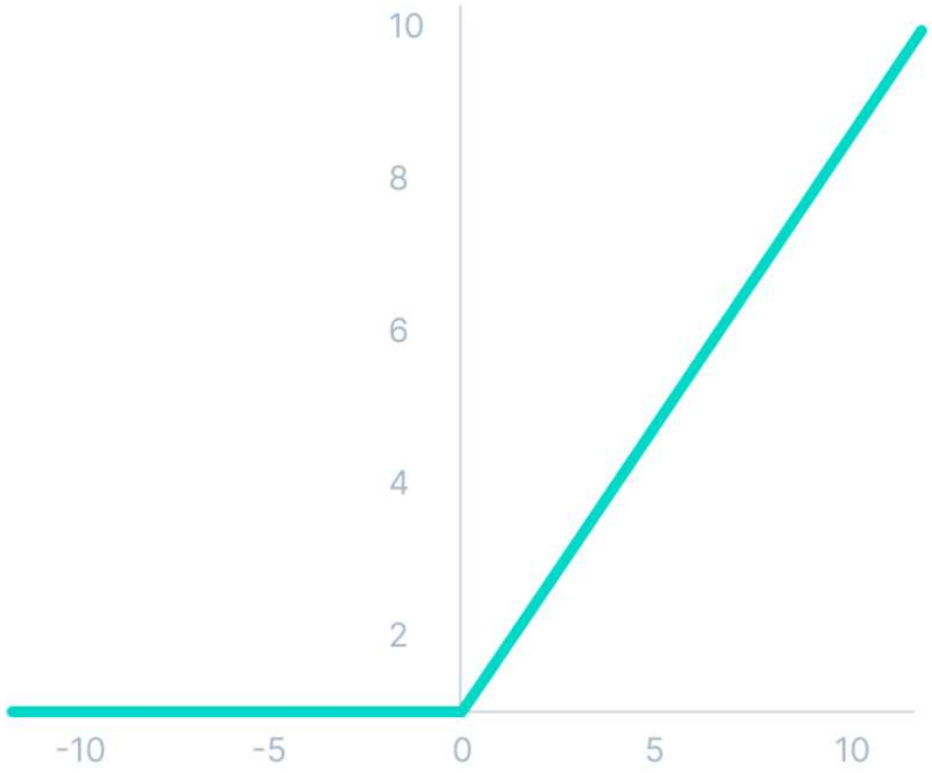

Figura 3.6: Grafico della ReLU

Questa funzione viene comunemente utilizzata come funzione di attivazione nelle CNN. Per quanto riguarda le funzioni di attivazione per l'output *layer* invece la più famosa (e una delle prima utilizzate dal punto di vista storico) è la sopracitata

[2] *Leaky relu.* URL: `https://www.v7labs.com/blog/neural-networks-activation-functions/`.

sigmoide visibile in Fig. 3.7. Questa funzione è utilizzata in qualsiasi situazione che richieda di esprimere un risultato con una probabilità in quanto il suo output è compreso tra 0 e 1, infatti è matematicamente descrivibile come:

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

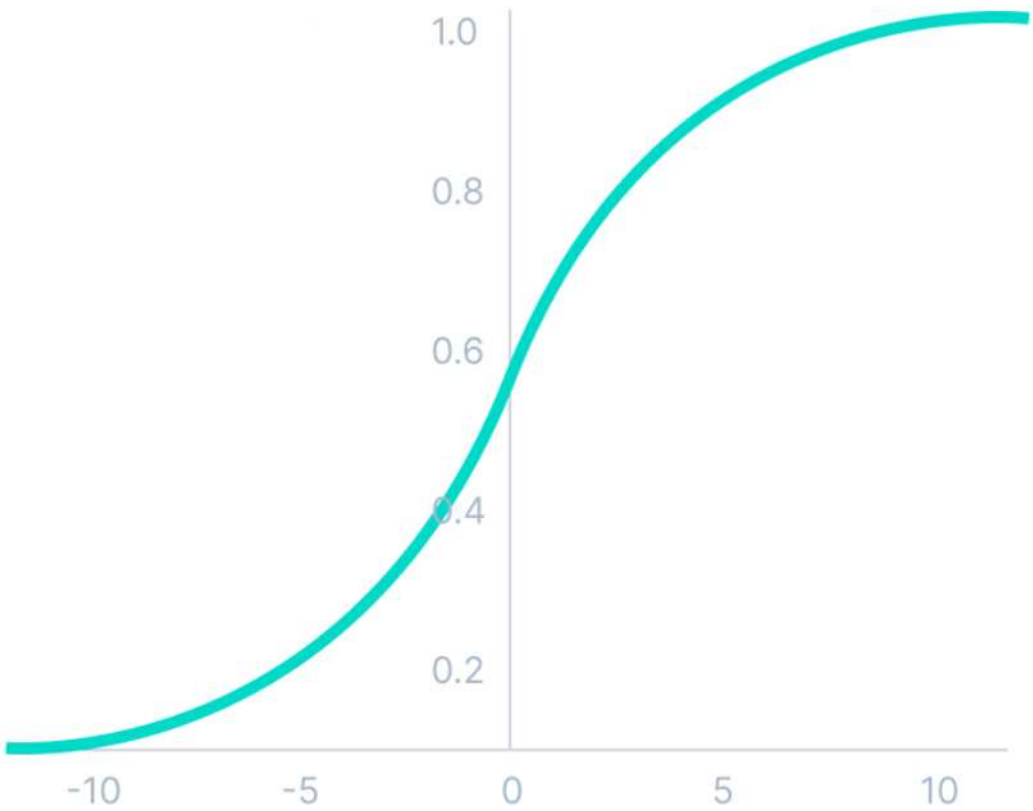

Figura 3.7: Grafico della sigmoide

Tuttavia è intuibile che questa funzione non sia applicabile a situazioni che prevedono la classificazione di elementi in 3 o più classi. Per ovviare a questo problema si possono combinare più sigmoidi in modo tale da ottenere la cosiddetta *softmax*, la quale restituisce le probabilità che un certo elemento appartenga ad ognuna delle classi disponibili, scegliendo poi il valore più alto per eseguire la predizione.

$$softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

Dove z è il vettore contenente tutti gli output della rete neurale e l' i-esimo valore di $softmax(z_i)$ è la probabilità che l'input appartenga alla classe i

3.1.2 Gradient descent

Il *gradient descent*[3] è un algoritmo di ottimizzazione che permette di trovare il minimo locale in una funzione differenziabile in modo iterativo e viene utilizzato per trovare i valori che permettono di minimizzare la funzione obiettivo tramite *backpropagation*. Il miglioramento dei valori viene misurato tramite un gradiente, se questo è pari a 0 il modello smette di apprendere. Il gradient descent è descrivibile come:

$$\theta_{k+1} = \theta_k - \eta \nabla J(\theta_k)$$

In questa formula θ_k rappresenta il vettore di pesi attuale mentre θ_{k+1} rappresenta il vettore di pesi dopo l'iterazione, $\eta >= 0$ è il passo della funzione (learning rate) e va selezionato con cura: troppo alto e si rischia di perdere il minimo, troppo basso e il costo computazionale diventa eccessivo (Fig. 3.8), infine possiamo notare ∇ che rappresenta il gradiente, ovvero il vettore che definisce la direzione e il verso in cui la funzione cresce più rapidamente (si può notare la presenza del segno - a indicare che ci stiamo muovendo nella direzione opposta). Infine J rappresenta la funzione di costo in relazione al k-esimo peso.

Figura 3.8: Confronto fra learning rate rispettivamente troppo grande, adeguato e troppo piccolo

[3] *opt.* URL: `https://builtin.com/data-science/gradient-descent`.

Esistono tre tipi di gradient descent:

Batch gradient descent: in questo caso l'errore per ogni esempio del *dataset* viene calcolato ma aggiornato solo dopo che sono stati calcolati tutti. Il vantaggio di questo approccio è la maggiore efficienza computazionale e la maggior stabilità del gradiente che però a volte non può raggiungere il miglior risultato possibile, tuttavia richiede che tutto il *training set* sia caricato in memoria contemporaneamente.

Stochastic gradient descent: in questo caso invece l'aggiornamento delle variabili è subito successivo alla valutazione di un dato del *dataset* di *training*: questo comporta una maggiore spesa computazionale e permette di seguire in modo dettagliato il tasso di miglioramento. Altro fattore importante è che questa maggior precisione è più soggetta a rumore.

Mini-batch gradient descent: questa strategia è una via di mezzo fra le due sopra descritte. Infatti il *dataset* di *training* viene suddiviso in parti più piccole (*batch*) in modo tale da combinare i vantaggi degli altri metodi. Per questo motivo è il metodo più utilizzato per allenare reti neurali nel capo del *deep learning*.

3.1.3 Ottimizzatori

Gli ottimizzatori sono algoritmi che modificano i pesi dei una rete neurale al fine di migliorare i risultati ottenuti da essa (in particolare ridurre la *loss* e migliorare l'*accuracy*).

RProp - *Resilient Propagation*[4,5]: RProp è un algoritmo di ottimizzazione basato sul *batch gradient descent* che aggiorna dinamicamente i valori dei singoli pesi basandosi, a differenza di altri algoritmi, solo sul segno (e non anche sull'intensità) del gradiente. Questa scelta è dovuta al fatto che funzioni diverse possano condividere punti di minimo pur avendo gradienti completa-

[4] *RProp*. URL: `https://florian.github.io/rprop/`.

[5] *RProp*. URL: `https://en.wikipedia.org/wiki/Rprop`.

mente diversi, il che porterebbe ad avere passi completamente diversi come mostrato nelle figure Fig. 3.9 e Fig. 3.10

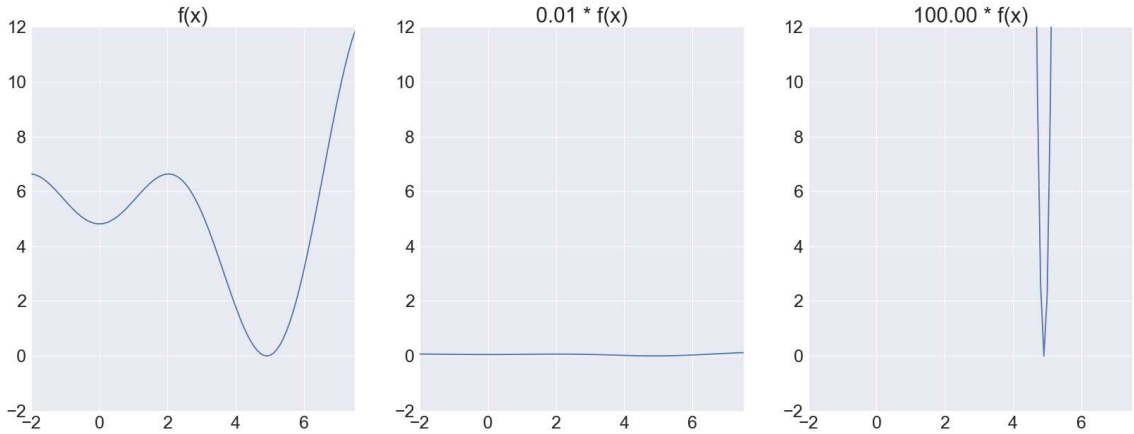

Figura 3.9: Esempi di funzioni con minimi uguali ma diversi gradienti

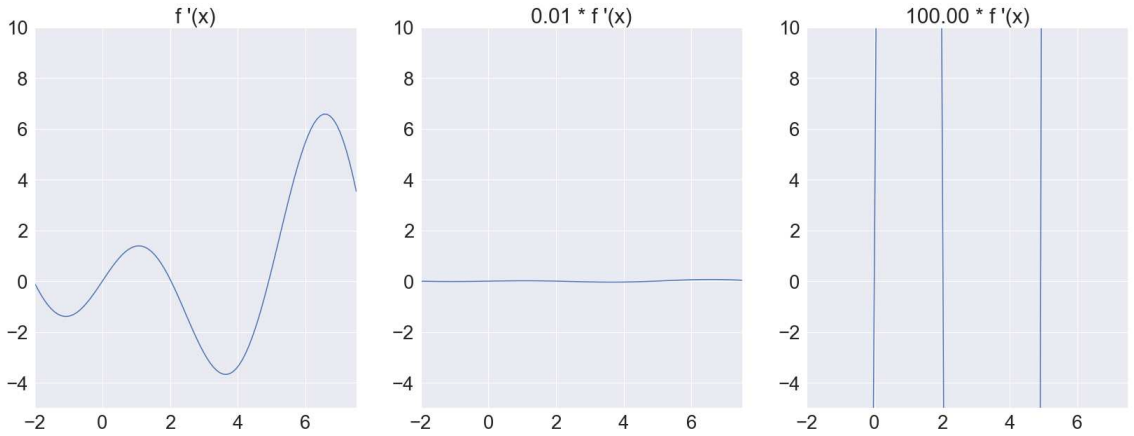

Figura 3.10: Confronto fra le derivate delle funzioni

L'aggiornamento dei pesi avviene secondo questa formula:

$$w_i^{(t)} = w_i^{(t-1)} - \eta_i^{t-1} sgn\left(\frac{\partial E^{t-1}}{\partial w_i^{t-1}}\right)$$

dove η_i^t è la dimensione del passo per la t-esima iterazione per l' i-esimo peso (η_i^1 e η_i^1 sono iperparametri da scegliere inizialmente) e $sgn\left(\frac{\partial E^{t-1}}{\partial w_i^{t-1}}\right)$ è il segno della derivata parziale dell'ultimo passo rispetto al relativo peso. Più concretamente il passo viene aggiornato confrontando il segno del gradiente attuale e della iterazione precedente: se il segno è rimasto uguale allora non si è ancora raggiunto il minimo e quindi si può incrementare la dimensione

del passo per raggiungere l'ottimo più rapidamente, mentre in caso contrario il punto di minimo è stato superato ed è quindi necessario ridurre il passo.

$$\eta_i^t = \begin{cases} min(\eta_i^{t-1} \times \alpha, \eta_{max}) & \text{se } \frac{\partial E^t}{\partial w_i^t} \times \frac{\partial E^{t-1}}{\partial w_i^{t-1}} > 0 \\ max(\eta_i^{t-1} \times \beta, \eta_{min}) & \text{se } \frac{\partial E^t}{\partial w_i^t} \times \frac{\partial E^{t-1}}{\partial w_i^{t-1}} < 0 \\ \eta_i^{t-1} & \text{altrimenti} \end{cases}$$

dove α e β servono per determinare se il passo vada aumentato o diminuito, mentre η_{max} e η_{min} evitano che il valore diventi troppo grande o troppo piccolo. Generalmente i valori per α e β sono 1,2 e 0,5 in modo tale da incrementare lentamente il passo e ridurlo velocemente. Inoltre si deve fare attenzione a non avere α e β uno il reciproco dell'altro, in modo tale da ottenere più valori diversi fra loro. Il problema principale di questo algoritmo è che i pesi possono aumentare e diminuire di molto in caso di gradienti molto forti.

AdaGrad[6,7]: algoritmo pensato per i casi in cui, a causa di dataset piccoli e con pochi esempi per certe feature rispetto ad altre che possono comparire più comunemente i pesi sono molto diversi fra loro (alcuni molto grandi e altri molto piccoli). La soluzione di questo algoritmo è analoga a quella di RProp e cioè adattare ogni singolo peso individualmente aumentando il passo nel caso in cui il parametro sia poco frequente e viceversa. Qui sotto possiamo vedere in ordine: la formula per l'aggiornamento dei pesi e come si calcola il passo all'iterazione t.

$$w_t = w_{t-1} - \eta'_t \frac{\partial L}{\partial w_{t-1}}$$

$$\eta'_t = \frac{\eta}{\sqrt{\alpha_t + \epsilon}}$$

dove η è un valore costante e ϵ è un piccolo valore positivo per impedire che il

[6] *AdaGrad*. URL: `https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827`.

[7] *AdaGrad*. URL: `https://towardsdatascience.com/understanding-deep-learning-optimizers-momentum-adagrad-rmsprop-adam-e311e377e9c2`.

denominatore sia zero, α_t invece si calcola così:

$$\alpha_t = \sum_{i=1}^{t} g_i^2$$

dove g_i è la derivata della *loss* rispetto al peso ed è uguale a:

$$g_i = \frac{\partial L}{\partial w(old)}$$

si può notare come g_i^2 sia un termine sempre positivo e quindi $\alpha_{t+1} >= \alpha_t$ e pertanto η'_t può solamente ridursi facendo sì che, con l'andare delle iterazioni, l'algoritmo imparerà più lentamente.

RMSProp - Root Mean Square Propagation[8,9]: algoritmo sviluppato per risolvere le problematiche dei due algoritmi sopra descritti, esso prevede di calcolare la media mobile dei gradienti elevati al quadrato in modo tale da prevenire la riduzione eccessiva del passo (problema di AdaGrad). Inoltre risulta più efficiente rispetto a RProp dato che diventa possibile usare il mini-batch gradient descent, cosa non possibile con RProp in quanto il passo sarebbe variato troppo intensamente (ad esempio con 10 mini-batch, 9 dei quali con un gradiente di 0.1 e uno con il gradiente di -0.9 il passo sarebbe stato aumentato 9 volte e diminuito una sola, sbilanciandolo di molto anche se ci si aspetterebbe che gli aumenti e le diminuzioni si bilanciassero). La formula per aggiornare i pesi è:

$$w_{i,j}(t) = w_{i,j}(t-1) - \frac{\eta}{\sqrt{E[h^2]_t + \epsilon}} \frac{\partial c}{\partial w_{i,j}}$$

$$E[g^2]_t = \beta E[g^2]_{t-1}$$

dove possiamo notare $E[g^2]_t$ la media mobile elevata al quadrato dei gradien-

[8] *RMSProp.* URL: `https://www.geeksforgeeks.org/rmsprop-optimizer-in-deep-learning/`.

[9] *RMSProp.* URL: `https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a`.

ti, $\frac{\partial c}{\partial w_{i,j}}$ il gradiente della funzione di costo rispetto al peso, η il passo e β parametro della media mobile comunemente impostato a 0.9.

Adam[10,11,12]: questo algoritmo, che si propone di combinare i vantaggi di AdaGrad e RMSProp, sfrutta il cosiddetto momento ovvero l'n-esimo valore previsto di una certa variabile all'n-esima iterazione, elevato alla n.

$$m_n = E[X^n]$$

Questo algoritmo utilizza due medie mobili m_t e v_t:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\mathrm{v}_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

qui possiamo vedere g_t il gradiente del *mini-batch* corrente, β_1 e β_2 i quali sono solitamente utilizzati con valori pari a 0.9 e 0.999 rispettivamente. Dato che queste medie sono stime dei momenti vogliamo che:

$$E[m_t] = E[g_t]$$

$$E[v_t] = E[g_t^2]$$

tuttavia ciò vorrebbe dire che le nostre stime non hanno bias cosa che non è vera,

[10] *Adam.* URL: `https://www.geeksforgeeks.org/adam-optimizer-in-tensorflow/`.

[11] *Adam.* URL: `https://it.wikipedia.org/wiki/Keras`.

[12] *Adam.* URL: `https://www.sciencedirect.com/science/article/abs/pii/S0020025522006260`.

infatti inizializzando m_t e v_t a 0 si può dimostrare come abbiano un bias verso lo 0:

$$m_0 = 0$$
$$m_1 = \beta_1 m_0 + (1 - \beta_1) g_1 = (1 - \beta_1) g_1$$
$$m_2 = \beta_1 m_1 + (1 - \beta_1) g_2 = \beta_1 (1 - \beta_1) g_1 + (1 - \beta_1) g_2$$

come si può vedere i valori dei gradienti iniziali hanno sempre meno valore. Tutto ciò si può riassumere con:

$$m_t = (1 - \beta_1) \sum_{i=0}^{t} \beta_1^{t-i} g_i$$

possiamo quindi correggere m_t e v_t tramite il processo di bias *correction*:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Grazie a questi parametri si ottiene la formula di aggiornamento dei pesi:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Il motivo per cui è stato scelto Adam come ottimizzatore è il basso costo di *training* e le alte *performance*.

3.2 Funzionamento CNN

3.2.1 Tensori

Per poter essere elaborate da una CNN le immagini devono essere opportunamente contenute all'interno di apposite strutture dati dette tensori. I tensori sono dei contenitori di informazioni, più tipicamente numeri, di dimensioni (anche dette assi) variabili. Altre caratteristiche dei tensori sono:

Rank: il numero di assi di un tensore

Shape: una tupla contenente il numero di dimensioni di ogni asse

Data type: il tipo di dato contenuto nel tensore: float32, uint8, float64... Raramente si possono trovare anche tensori di tipo char

Un esempio pratico di tensore è una matrice con 3 colonne e 5 righe: essa possiede due assi (*rank* 2) e una *shape* pari a (3,5). Generalmente un' immagine a colori è caratterizzata da un altezza, una larghezza e la profondità dei colori, pertanto si possono memorizzare all'interno di tensori 3D come ad esempio (128,28,28,3) dove 128 è la dimensione del *batch* di immagini, 28 e 28 sono le dimensioni delle immagini e infine 3 è la profondità dei colori per un'immagine RGB (per un'immagine in bianco e nero dovrebbe essere 1). In Fig. 3.11 e Fig. 3.12 si possono vedere un'immagine in bianco e nero la relativa matrice 28x28, prese dal dataset MNIST[13].

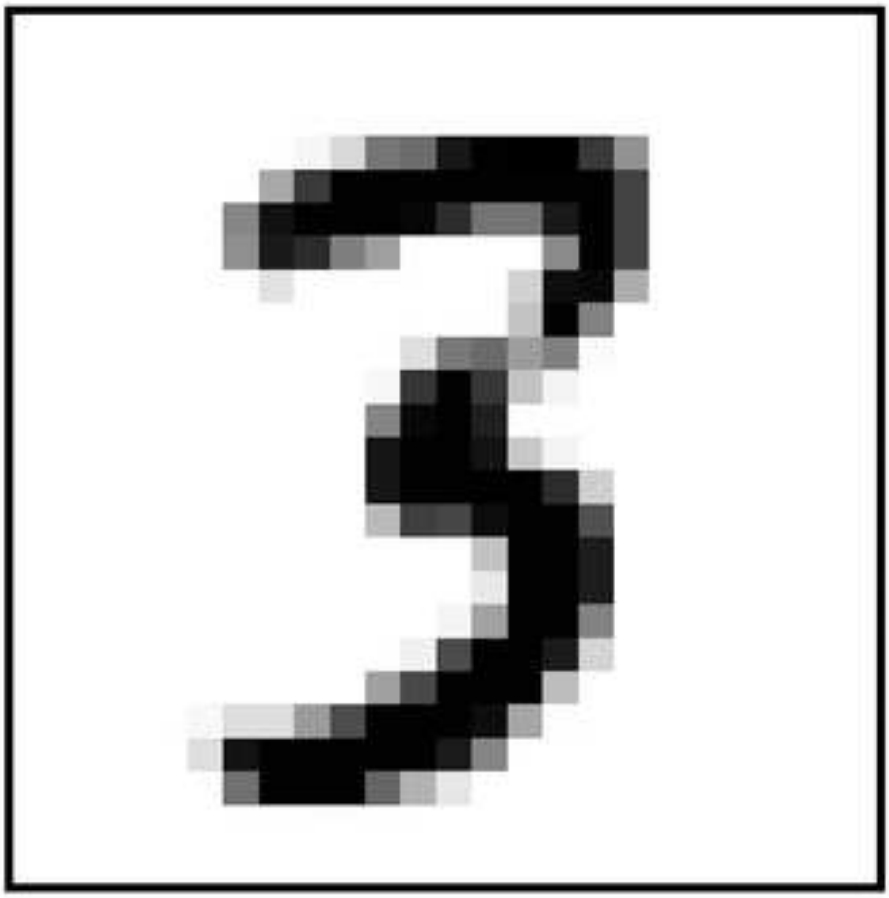

Figura 3.11: Esempio di un immagine

[13]Rita Fioresi e F. Zanchetta. *Deep Learning and Geometric Deep Learning: an introduction for mathematicians and physicists*. Mag. 2023. DOI: 10.48550/arXiv.2305.05601.

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	8	38	137	146	232	254	255	255	197	109	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	87	197	253	253	253	253	253	253	253	253	188	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	120	237	253	253	253	248	209	139	139	230	253	188	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	112	229	210	128	96	0	0	0	0	117	253	188	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	28	0	0	0	0	0	0	45	241	245	82	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	60	253	125	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	32	134	148	98	127	4	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	7	201	253	200	59	12	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	122	246	253	223	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	233	253	253	236	55	11	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	233	253	253	253	253	210	48	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	68	193	185	243	253	253	173	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	59	253	253	226	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	22	253	253	226	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	93	253	253	123	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	14	180	253	253	228	42	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	95	181	253	253	253	66	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	6	33	33	100	178	253	253	253	241	88	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	32	236	253	253	253	253	253	228	122	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	143	253	253	253	154	76	24	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 3.12: Esempio di una matrice

3.3 Layer CNN

Come detto precedentemente tra input e output *layer* sono presenti gli *hidden layer*, ovvero gli strati preposti all'elaborazione degli input. Di questi ne esistono varie tipologie.

3.3.1 Layer convoluzionali

Questi *layer* si occupano di imparare a rilevare specifiche *feature* dei dati forniti contenute in spazi bidimensionali all'interno delle immagini dette filtri o *kernel*. Una volta appresa una *feature* essa può essere individuata ovunque essa appaia nell'immagine (a differenza di ciò che succede con le feature apprese da un *dense layer*, che sono rilevabili solo nella zona in cui sono state apprese) permettendo di avere un risparmio sulla quantità di dati da fornire in input. Inoltre il processo di convoluzione permette anche di apprendere le gerarchie spaziali tra le diverse *feature* (per esempio se la rete è allenata nell'identificazione di gatti probabilmente apprenderà la posizione delle orecchie o degli occhi rispetto al resto del muso). Questo processo utilizza tensori 3D, detti *feature maps*, in cui i tre assi rappresentano rispettivamente larghezza e altezza dell'immagine, che possono variare tra i diversi *layer*, e

il numero di canali, i quali rappresentano la quantità di informazione presente nel *layer* precedente. Un *layer* di input per un'immagine RGB, ad esempio, espone tre canali, uno per ogni colore, mentre un *layer* convoluzionale espone una quantità arbitraria di canali, uno per ogni *feature* appresa. Nell'esempio precedente una *feature* potrebbe codificare la forma dell'orecchio. Durante il processo di apprendimento dei filtri (o *kernel*), di dimensioni variabili, tipicamente 3×3 oppure 5×5, vengono fatti scorrere sull'immagine ottenendo varie sezioni 3D delle varie *feature* le quali vengono trasformate in vettori 1D che sono successivamente riassemblati in output map 3D. Come detto precedentemente possono esserci delle variazioni di altezza e larghezza tra input *feature maps* e output *feature maps*, ciò può avere due cause:

Effetti di bordo: se si considera una *feature map* di dimensione 3×3 che si muove su uno spazio 5×5 come quella in Fig. 3.13 si può notare come essa si possa spostare senza fuoriuscire mantenendo il centro solo in una delle 9 posizioni centrali: è quindi evidente come l'immagine si riduca ad un 3×3.

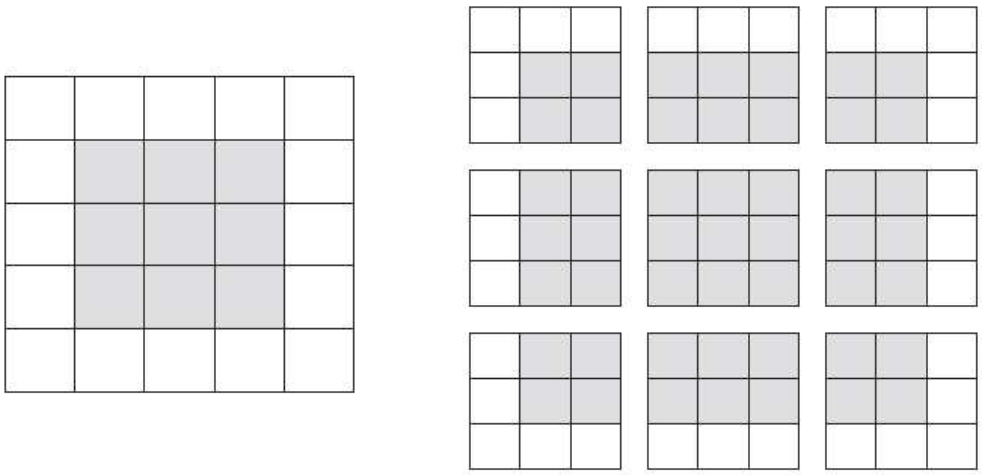

Figura 3.13: Esempio effetto di bordo

Una possibile soluzione a questo problema è il *padding* e cioè l'inserimento di una quantità di colonne e righe sufficienti a rendere anche i bordi posizioni valide.

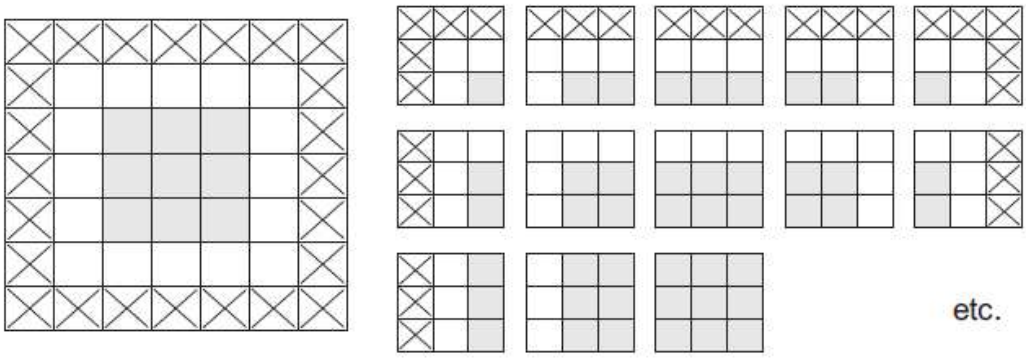

Figura 3.14: Esempio utilizzo *padding*

Uso degli *stride*: Questa tecnica, raramente utilizzata, serve a sottocampionare un input prevede che i centri delle varie *feature maps* non siano contigui ma abbiano degli spazi intermedi detti appunto *stride*. Una sua applicazione si può vedere in Fig. 3.15

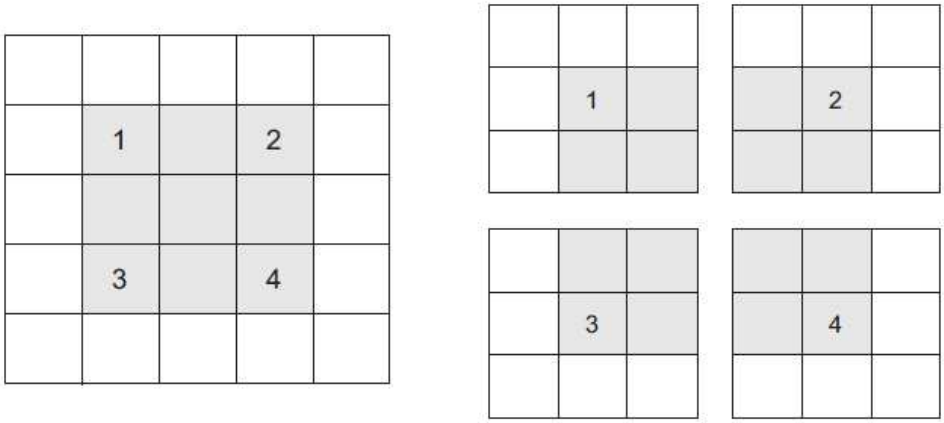

Figura 3.15: Esempio utilizzo stride

3.3.2 Pooling layer

Ulteriore causa delle variazioni di dimensioni degli input dopo l'elaborazione è dovuta a questo tipo di *layer* che si occupa di fare sottocampionamento degli input al fine di ridurre la quantità di parametri da apprendere aumentando così la velocità di calcolo. Questo processo rende anche il modello più robusto perché lega tra loro le *feature* che riceve dal *layer* convoluzionale. Ne esistono 3 tipi:

Max pooling: in questo caso l'operazione di pooling seleziona solamente il valore più alto (ovvero la *feature* di maggior impatto) della *feature map*. Questa è stata la tipologia implementata nelle CNN create in questo stage in quanto generalmente genera risultati migliori rispetto alle alternative (è più sensato e utile osservare le feature più prominenti invece che eseguire una media che considera anche feature poco rilevanti).

Average pooling: in questo caso invece l'operazione di *pooling* calcola la media di tutti i valori interessati dalla *feature map*.

Global pooling: questa modalità di *pooling* (che si può specializzare ulteriormente in *gloabal max pooling* o *global average pooling*) ottiene un unico valore a partire da tutti i valori di un canale.

3.3.3 Dropout layer

Al fine di evitare l'*overfitting* dovuto al modello che impara a riconoscere il rumore del *dataset* di *training* è possibile introdurre questa tipologia di *layer*, la quale disattiva con una certa probabilità alcuni neuroni scelti in modo casuale. Infatti nel caso in cui venissero mantenuti potrebbero rimediare agli errori degli altri adattandosi a vicenda. Matematicamente questo *layer* modifica la *forward propagation* così:

$$\begin{cases} r_j^l \sim Bernoulli(p) \\ \tilde{y} = r^{(l)} \times y^{(l)} \\ z_i^{(l+1)} = w_i^{(l+1)} \tilde{y}_l + b_i^{l+1} \\ y_i^{(l+1)} = f(z_i^{(l+1)}) \end{cases}$$

dove $z_i^{(l+1)}$ è il vettore di output del *layer* l+1 prima dell'attivazione, $y_i^{(l+1)}$ è il vettore di output del *layer* l, $w_i^{(l+1)}$ è il peso del layer l e b_i^{l+1} il suo bias. Prima di calcolare z ogni *layer* è campionato e moltiplicato con le variabili indipendenti di Bernoulli, rappresentate da r_j^l.

3.3.4 Batch Normalization layer

È un tipo di *layer* utilizzato per stabilizzare i pesi durante il *training*, migliorando le *performance*, dato che, secondo alcune correnti di pensiero contrasta il fenomeno dell' *Internal Covariate Shift* (ICS). Questo fenomeno è causato dalla variazione delle distribuzioni dei pesi nei *layer* più interni in risposta alle variazioni casuali degli input dei *layer* più esterni. L'intensità di queste variazioni viene amplificata dalla profondità del modello, risultando un problema particolarmente legato al *deep learning*. Questo tipo di *layer* inoltre alza il *learning rate* eliminando casi di svanimento o esplosione del gradiente. Inoltre dovrebbe ridurre l'*overfitting* aumentando le capacità di generalizzazione del modello (rendendo inutile il *dropout*). Come detto questa è una corrente di pensiero che recenti studi parrebbero aver di-

screditato suggerendo che in realtà questo *layer* migliori la *loss function* rendendo i suoi cambiamenti di gradiente meno repentini[14].

3.3.5 Flatten

Questo tipo di *layer* è utilizzato per trasformare tensori multidimensionali in vettori unidimensionali, riducendo così la dimensionalità dei dati. Ciò è necessario per utilizzare gli output dei *layer* di *pooling* e convoluzionali come input per i *dense layer*, che richiedono un formato vettoriale.

3.3.6 Dense

I *layer* di questo tipo sono caratterizzati dal fatto che ogni neurone di un *layer* è connesso ad ogni neurone di quello precedente. L'input di ogni neurone è pari a:

$$z_j = \sum_i (w_{ij} \times x_i) + b_j$$

Dove w_{ij} è il peso proveniente dal neurone i del precedente *layer* verso il neurone j, x_i è l'input proveniente dal neurone i e b_j è il bias applicato al neurone j. Il risultato di questo calcolo viene dato in input alla funzione di attivazione che, introducendo la non linearità, permette l'apprendimento di funzioni complesse.

Generalmente, all'interno di una CNN, vengono utilizzati come ultimo *layer* in modo tale da interpretare tutte le *feature map* generate e fornire le predizioni come output.

[14] Shibani Santurkar et al. *How Does Batch Normalization Help Optimization?* 2019. arXiv: `1805.11604 [stat.ML]`. URL: `https://arxiv.org/abs/1805.11604`.

Capitolo 4

Descrizione dello stage

4.1 Introduzione al progetto

Lo stage si basa sulla necessità di creare un'intelligenza artificiale in grado di riconoscere alcune caratteristiche delle provette inserite in una macchina, la GENEQUALITY® TWENTYFIFTY. Quest'ultima, fornita da AB Analitica, si occupa di eseguire in modo autonomo vari tipi di analisi, come ad esempio la PCR, relative a fluidi corporei, contenuti in tipi diversi di provette. Per fare ciò deve riconoscere la presenza o l'assenza delle provette sul *deck*, le loro dimensioni (suddividendo le provette nelle varie categorie) in modo tale da regolare la velocità di aspirazione e se esse siano pronte all'estrazione dei fluidi tramite pipette automatizzate (in particolare il sistema deve segnalare la presenza di tappi sulle provette, i quali danneggerebbero la macchina). A questo fine si sono dovuti implementare 3 diversi algoritmi di intelligenza artificiale basati sulle reti convoluzionali: uno per individuare le provette dei *rack* primari, il quale contiene il numero maggiore di possibili variazioni delle provette, distribuito sul numero maggiore di classi, uno per individuare le provette dei *rack* secondari, i quali conentngono un tipo unico di provette non utilizzabile sull'altra tipologia di rack, e uno per individuare una casistica precisa relativa ai campioni ausiliari, che sfrutta alcune delle provette utilizzate nei rack primari. Inoltre è stato richiesto di verificare se fosse possibile addestrare un quarto modello al fine di riconoscere quando il cestino interno alla macchina fosse da svuotare. L'utilizzo delle reti neurali si è rivelato necessario in quanto la varietà di possibili

situazioni e caratteristiche che le provette possono avere è molto ampia e pertanto non è pensabile di poter, ad esempio, identificare il colore rosso per individuare un tappo in quanto esistono molti tipi di tappi diversi con colorazioni diverse in condizioni di luce e posizioni diverse.

4.2 Analisi preventiva dei rischi

Durante la fase di analisi iniziale sono stati individuati alcuni possibili rischi a cui si potrà andare incontro. Si è quindi proceduto a elaborare delle possibili soluzioni per far fronte a tali rischi.

1. Errore di classificazione delle provette

Descrizione: durante la creazione del *dataset*, esiste il rischio di errore umano nella classificazione delle provette, a causa delle piccole variazioni dimensionali che possono rendere difficile distinguere correttamente le immagini..

Soluzione: coinvolgimento del responsabile a capo del progetto per ottenere le specifiche da parte di AB analitica.

2. Errore di classificazione dovuto a condizioni estreme di luce

Descrizione: la rete neurale non è in grado di riconoscere correttamente le provette se sottoposte a condizioni di luce troppo intensa o troppo debole.

Soluzione: aumento della quantità di immagini ottenute in queste condizioni, coprendo fonti di luce preesistenti oppure aumentandole il più possibile.

3. Errore di classificazione dovuto alla sovrapposizione di provette di dimensioni diverse

Descrizione: la rete neurale non è in grado di riconoscere le provette se coperte da provette di dimensioni maggiori.

Soluzione: aumento della quantità di immagini ottenute in queste situazioni, sfruttando anche posizioni anomale dei tappi al fine di coprire aree maggiori.

4. Errore di classificazione dovuto al posizionamento errato del *rack*

Descrizione: la rete neurale non è in grado di riconoscere le provette se il *rack* non è posizionato correttamente.

Soluzione: aumento della quantità di immagini ottenute con il *rack* leggermente spostato nella corsia oppure inserito nella corsia sbagliata.

5. Errore di classificazione dovuto alla presenza di liquidi all'interno delle provette

Descrizione: la rete neurale non è in grado di riconoscere le provette se contengono liquidi a causa dei colori e dei riflessi prodotti dai liquidi.

Soluzione: aumento della quantità di immagini ottenute con le provette che contengono liquidi.

6. Difficoltà condivisione macchina

Descrizione: è molto probabile che nello stesso lasso temporale più persone debbano utilizzare la macchina per *task* diverse e possibilmente conflittuali (test e installazione di nuovi componenti ad esempio rendono inservibile la macchina).

Soluzione: coordinazione con il team durante gli *stand up meeting* giornalieri.

7. Documentazione insufficiente o deprecata

Descrizione: anche a causa di problemi *hardware* è stato necessario utilizzare versioni di librerie e tecnologie per le quali è difficile individuare una documentazione e quindi trovare un supporto adeguato al fine di verificare le compatibilità.

Soluzione: ricerche approfondite per la ricerca di soluzioni.

Capitolo 5

Analisi dei requisiti

5.1 Tracciamento dei requisiti

Da un'attenta analisi dei requisiti e degli use case effettuata sul progetto è stata stilata la tabella che traccia i requisiti in rapporto agli use case.
Sono stati individuati diversi tipi di requisiti e si è quindi fatto utilizzo di un codice identificativo per distinguerli.
Il codice dei requisiti, dove ogni requisito è identificato con il carattere **R**, è composto da tipologie di vincolo e livello di importanza del vincolo. In particolare le tipologie di vincolo sono:

F: Funzionale.

Q: Qualitativo.

V: Di vincolo.

Mentre i livelli di importanza del vincolo sono:

O: Obbligatorio (necessario).

D: Desiderabile.

Z: Opzionale.

Nelle tabelle 5.1, 5.2 e 5.3 sono riassunti i requisiti e il loro tracciamento con gli use case delineati in fase di analisi.

5.2 Tabelle dei requisiti

Requisito	Descrizione	Stato
RFO-1	Il sistema è in grado di identificare la presenza di una provetta con tappo nel *rack* principale	Soddisfatto
RFO-2	Il sistema è in grado di identificare la presenza di una provetta grande nel *rack* principale	Soddisfatto
RFO-3	Il sistema è in grado di identificare la presenza di una provetta media nel *rack* principale	Soddisfatto
RFO-4	Il sistema è in grado di identificare la presenza di una provetta piccola nel *rack* principale	Soddisfatto
RFO-5	Il sistema è in grado di identificare l'assenza di una provetta nel *rack* principale	Soddisfatto
RFD-6	Il sistema è in grado di identificare se il cestino è vuoto o pieno	Soddisfatto
RFO-7	Il sistema è in grado di identificare la presenza di una provetta aperta nel *rack* ausiliario	Soddisfatto
RFO-8	Il sistema è in grado di identificare la presenza di una provetta con il tappo nel *rack* ausiliario	Soddisfatto
RFO-9	Il sistema è in grado di identificare l'assenza di una provetta nel *rack* ausiliario	Soddisfatto

Tabella 5.1: Tabella del tracciamento dei requisiti funzionali.

Requisito	Descrizione	Stato
RQO-1	Il riconoscimento delle varie categorie deve avvenire indipendentemente dalle condizioni di luce	Soddisfatto
RQD-2	Il riconoscimento delle varie categorie deve avvenire anche in condizioni anomale di posizionamento delle provette	Soddisfatto
RQD-3	Il processo di riconoscimento deve avere una durata inferiore a 1 secondo	Soddisfatto

Tabella 5.2: Tabella del tracciamento dei requisiti qualitativi.

Requisito	Descrizione	Stato
RVO-1	Le immagini usate per l'addestramento devono essere correttamente etichettate	Soddisfatto

Tabella 5.3: Tabella del tracciamento dei requisiti di vincolo.

Capitolo 6

Codifica

6.1 Tecnologie e strumenti

Di seguito viene data una panoramica delle tecnologie e strumenti utilizzati. In Fig. 6.1 sono presenti i rispettivi loghi.

Ambiente di sviluppo

L'ambiente di sviluppo utilizzato è Visual Studio Code, utilizzato tramite WSL 2 (Windows Subsystem for Linux) il quale ha permesso di poter sfruttare la GPU (fondamentale per ridurre drasticamente i periodi di training dei modelli sviluppati) nonostante il supporto ad essa relativo da parte delle librerie utilizzate (in particolare TensorFlow) sia stato interrotto nella versione attuale di Windows nativo. Il linguaggio utilizzato è Python.

Keras[1] e TensorFlow[2]

Keras è una libreria *open source* per l'apprendimento automatico e le reti neurali, scritta in Python. È progettata come un'interfaccia a un livello di astrazione superiore di altre librerie simili di più basso livello, e supporta come *back-end* TensorFlow, popolare libreria open source per l'apprendimento automatico, sviluppata da Google. Essa viene utilizzata per formare e distribuire modelli di *machine lear-*

[2] *Tensorflow.* URL: `https://it.wikipedia.org/wiki/TensorFlow`.

ning di vario tipo, come ad esempio quelli di *computer vision* utilizzati in questo stage. TensorFlow infatti consente di specificare l'architettura di un modello, incluso il numero di livelli, il tipo di funzione di attivazione e l'ottimizzatore da utilizzare, quindi fornisce gli strumenti per addestrare il modello su un set di dati. Uno dei punti di forza di TensorFlow è la sua scalabilità, che consente di addestrare modelli su un'ampia gamma di *hardware*, da una singola CPU a più *cluster* distribuiti.

Matplotlib

Matplotlib è una libreria per la creazione di grafici per il linguaggio di programmazione Python e la libreria matematica NumPy. Grazie ad essa è stato possibile visualizzare le metriche dei modelli e visualizzare gli errori incontrati in modo tale da correggerli in modo mirato (ad esempio visualizzando le immagini classificate in modo errato alla fine di un training in modo tale da integrare più immagini simili ad esse all'interno del dataset).

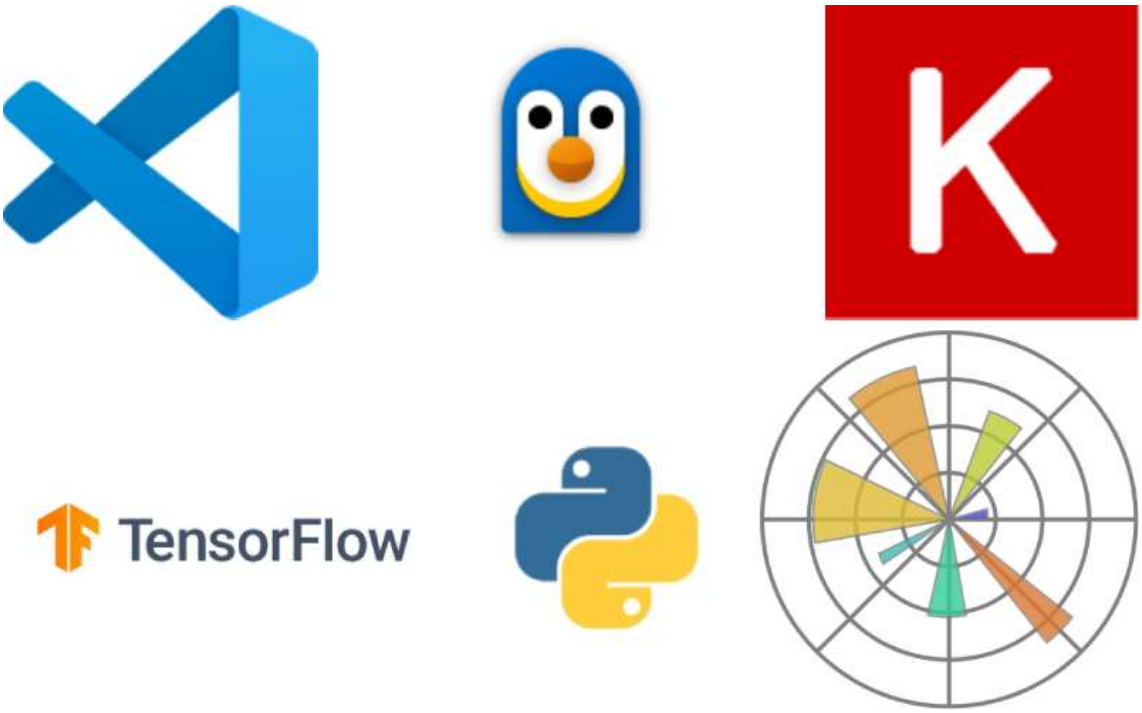

Figura 6.1: Loghi delle varie tecnologie usate

Altre tecnologie

Durante lo sviluppo del progetto e la stesura della tesi sono state utilizzate anche altre tecnologie come miniconda (*installer* di conda, un *package manger open source*) e Overleaf (editor di testo utilizzato per scrivere la tesi in LaTeX). Inoltre i modelli sono stati convertiti in TensorFlow Lite su richiesta dell'azienda, affinché possano essere integrati in altri flussi di lavoro scritti in Go come parte di uno sviluppo futuro.

6.2 Progettazione

In questo capitolo viene descritto il processo di creazione dei vari dataset e la struttura delle reti neurali sviluppate.

6.2.1 Introduzione

Larga parte dello stage è stata dedicata all'ampliamento dei dataset preesistenti e alla creazione di nuovi dataset. Per fare ciò è stato necessario utilizzare la macchina fornita da AB analitica.

6.2.2 GENEQUALITY® TWENTYFIFTY (GQ 2050)

Il GENEQUALITY® TWENTYFIFTY (GQ 2050) è un sistema automatizzato *sample-to-result*, con termociclatore integrato, per l'estrazione di acidi nucleici da campioni biologici, la preparazione e la corsa di reazioni PCR Real time e il *reporting* dei risultati. È dotato inoltre di 4 corsie dove si possono inserire dei *rack*, supporti appositi per mantenere in posizione le provette. Esistono 2 tipi di *rack*: primari e ausiliari. I primi sono dotati di 16 spazi di uguali dimensioni dove si possono inserire provette di varie dimensioni (dagli 11 ai 16 mm di diametro e dai 65 ai 105 mm di altezza), i secondi invece sono dotati di 24 spazi i quali sono così suddivisi: il primo e l'ultimo sono delle stesse dimensioni degli spazi dei *rack* primari e sono pensati per provette di dimensione grande, mentre gli altri 22 sono di dimensioni più ridotte possono essere utilizzati solo per una categoria a se stante di provette. Per i *rack*

primari si è addestrato un algoritmo mentre per i *rack* ausiliari è stato necessario addestrare due algoritmi diversi per individuare i due diversi tipi di slot. È inoltre presente un cestino all'interno del quale vengono buttate delle punte monouso per l'estrazione dei fluidi. Inizialmente durante la raccolta dei dati si è dovuto tenere conto delle posizioni corrette dei diversi *rack* (partendo da sinistra infatti si disponevano *rack* ausiliario e due *rack* primari come si vede in Fig. 6.2) facendo attenzione che i rack fossero posizionati correttamente in fondo alla corsia. In seguito, sia per ampliare il *dataset* che per aumentare la robustezza dell'algoritmo, si sono spostati i *rack* in posizioni anomale, inserendo anche degli spessori sul retro in modo tale da non poterli spingere fino in fondo per far si di imitare la struttura di un altro prototipo della macchina con una struttura leggermente diversa. Sempre nella Fig.6.2 si può vedere il cestino in alto a sinistra il quale si riempie con le punte nere visibili in basso a destra.

Figura 6.2: Immagine di calibrazione per il riconoscimento delle provette

6.2.3 Dataset

Per poter allenare le CNN era fondamentale ottenere un dataset sufficientemente ampio e vario da coprire equamente tutte le casistiche possibili. Pertanto una sostanziale parte del tempo è stata dedicata a scattare fotografie alle provette le quali, dopo ogni foto, andavano spostate in modo casuale in modo tale da ottenere esempi

di ogni tipologia di provette nelle varie posizioni, nelle varie condizioni di luce (in particolare in condizioni di riflessi molto intensi i quali rendono difficile le classificazione come mostrato in Fig. 6.3) e anche ruotate in modo diverso per rendere visibili diverse sezioni delle etichette.

Figura 6.3: A sinistra le provette senza riflesso, a destra provette dello stesso tipo con riflesso

Altro fattore da notare è l'interazione tra le diverse provette: ad esempio una provetta grande posta negli slot vicini alle prime o alle ultime posizioni può coprire parzialmente o totalmente una provetta piccola (Fig. 6.4) oppure un insieme di provette potrebbe creare un gioco di ombre sulle provette circostanti, falsando così i risultati. Infatti la luce è un elemento molto importante, al punto che la macchina è provvista di una porta in grado di schermare la luce a meno che essa non sia troppo forte: pertanto il dataset includeva in egual misura foto fatte con la porta aperta e foto con la porta chiusa, in diversi momenti della giornata e con le finestre della stanza aperte e chiuse. Infine, per rendere migliore la classificazione dei tappi, essi sono stati inseriti anche in posizioni anomale (Fig. 6.4). Sono inoltre presenti vari tipi di provette anche all'interno delle stesse categorie come si può vedere in Fig. 6.5

Figura 6.4: A sinistra tappi in posizioni anomale, a destra provette parzialmente coperte da altre provette

Figura 6.5: Esempi di diverse provette appartenenti alla stessa classe (tappi), in basso a destra un tappo del *rack* ausiliario

Una volta ottenuto un certo numero di immagini era necessario classificarle preparando dei file csv come quello mostrato in Fig. 6.6, i quali rispecchiavano le posizioni delle provette. Si ottenevano così delle coppie foto-file csv con lo stesso nome. Per comodità si scriveva un unico file per "set" di foto per poi suddividerli con uno script python.

notlabeled	empty	large	notlabeled
notlabeled	empty	empty	notlabeled
notlabeled	empty	empty	notlabeled
notlabeled	empty	empty	notlabeled
notlabeled	empty	empty	notlabeled
notlabeled	large	empty	notlabeled
notlabeled	empty	empty	notlabeled
notlabeled	small	large	notlabeled
notlabeled	large	empty	notlabeled
notlabeled	empty	large	notlabeled
notlabeled	empty	empty	notlabeled
notlabeled	empty	large	notlabeled
notlabeled	small	cap	notlabeled
notlabeled	empty	empty	notlabeled
notlabeled	empty	empty	notlabeled
notlabeled	small	cap	notlabeled

Figura 6.6: Esempio di un file csv per il *rack* primario, le colonne notlabeled non vengono considerate al momento del ritaglio e indicano dei *rack* mancanti

Una volta completati i file csv era possibile utilizzare uno script python per poter ritagliare correttamente le immagini: usando le immagini di calibrazione come quella mostrata in Fig. 6.2 la macchina poteva ricavare i centri delle provette i quali venivano usati per ritagliare delle aree (nel caso dei *rack* primari quadratati 100×100 px, nel caso degli ausiliari quadrati 60×60px) che includevano singole provette. Questi ritagli venivano poi automaticamente rinominati utilizzando il numero della foto, dello slot e la classificazione data nel file csv per quella posizione. In seguito è stato anche aggiunto un ulteriore passaggio che duplicava di fatto il *dataset* di ritagli andando a introdurre uno spostamento di massimo 10 px casuale nel momento del ritaglio. Questi ritagli vengono anche automaticamente smistati in cartelle, una per classe. Infine con un altro script il *dataset* viene suddiviso in 3 categorie, 2 delle quali mantengono la distinzione fra le classi ovvero il *training set*, il *validation set* e il *test set*.

6.3 Codifica

6.3.1 Introduzione

Come detto le CNN sviluppate sono 4 e presentano strutture simili ma con alcune differenze.

6.3.2 CNN primari

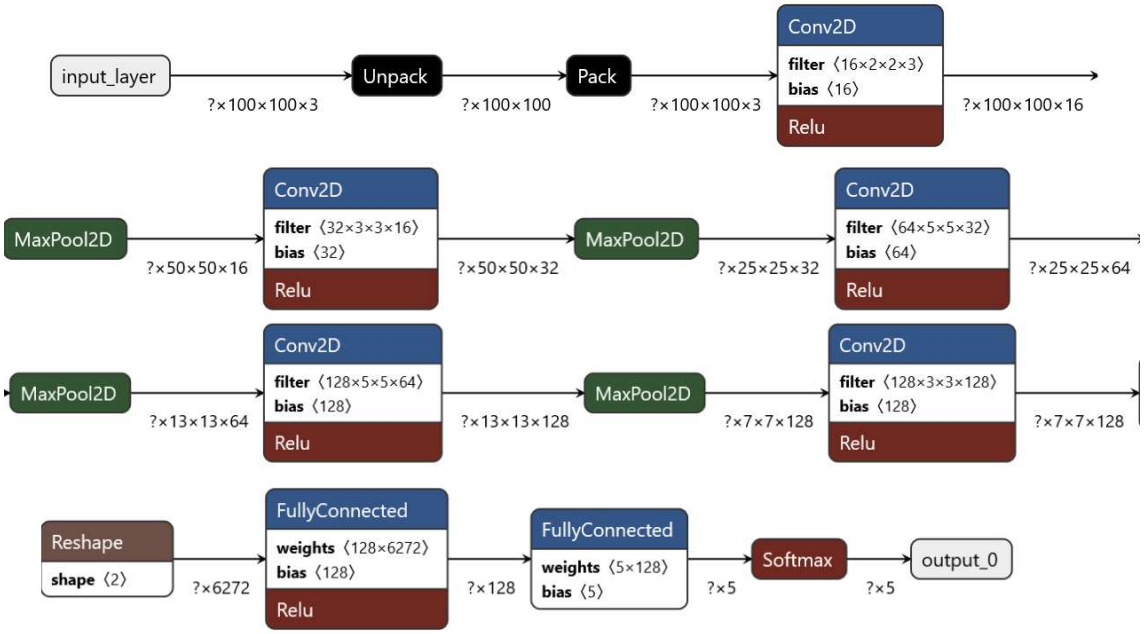

Figura 6.7: Esempio della struttura di uno dei modelli usati

I componenti di questa prima CNN sono vari e saranno d'esempio per la trattazione delle altre reti neurali sviluppate. Per prima cosa si vede il *layer* di input il quale richiede una *shape* pari a (100,100,3): i primi due valori indicano altezza e larghezza dell'immagine in pixel mentre il terzo parametro indica il numero di canali da dedicare al colore (3 per un'immagine RGB).

A questo punto c'è il primo layer convoluzionale il quale applica 16 filtri di dimensione 2×2, applicando un *padding* che viene utilizzato per evitare che le informazioni ai bordi dell'immagine vengano tagliate durante l'applicazione dei filtri, permettendo così di mantenere la dimensione originale. Infine viene determinata la funzione di attivazione del *layer*, la relu. A seguire troviamo un *layer* di max pooling

il cui unico parametro imposta un *padding* analogo a quello appena descritto. Da qui i *layer* convoluzionali e di max pooling si alternano con un crescente numero di filtri di dimensioni sempre maggiori in modo da catturare *feature* più generali per poi ridurre la dimensione dei *kernel* nell'ultimo strato così da "compattare" le informazioni ottenute. Ora si trova un *layer flatten* in preparazione a due *layer dense* intervallati da un *layer* di *dropout*. L'ultimo layer dense cambia funzione di attivazione essendo incaricato di calcolare le probabilità con cui ogni elemento appartiene a ciascuna classe ed effettuare le predizioni. Il numero delle classi è indicato dal parametro precedente alla funzione di attivazione.

```
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['CategoricalAccuracy'])
batch_size = 30
```

In questo frammento di codice possiamo vedere la selezione della funzione di *loss*, dell'ottimizzatore e della metrica da calcolare. Inoltre è presente anche il numero di elementi che compone un batch.

```
stop_callback = keras.callbacks.
              EarlyStopping(monitor='val_loss',patience=14)
checkpoint_path = './model.keras'

checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
    filepath=checkpoint_path,
    save_weights_only=False,
    monitor='val_CategoricalAccuracy',
    mode='max',
    save_best_only=True
  )
```

Qui possiamo vedere la definizione dei *callbacks* ovvero funzioni che vengono eseguite in momenti specifici del *training*. In particolare *stop_callback* viene invocato, fermando il training, quando la metrica monitorata (in questo caso la *validation loss*) smette di migliorare entro un certo lasso di tempo (14 epoch nell'esempio). La

seconda *callaback* invece viene invocata al termine di ogni *epoch* e ha lo scopo di massimizzare la metrica monitorata, la *categorical accuracy* della validazione. Per fare ciò la *callback* salva l'intero modello (e non solo i pesi delle variabili) ogni volta che la metrica monitorata supera il valore massimo ottenuto nelle *epoch* precedenti.

```
train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=40,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True,
        vertical_flip=True,
        fill_mode='nearest',
        )
```

Questa sezione di codice è deputata al *data augmentation* ovvero la tecnica che permette di aumentare artificialmente le dimensioni del dataset alterando i dati preesistenti. In particolare, nel caso delle immagini e del codice proposto, le immagini vengono sottoposte a trasformazioni di intensità casuale come rotazioni, spostamenti, zoom, capovolgimenti verticali e orizzontali. Questa tecnica si utilizza per vari motivi in particolare:

***Dataset* ridotto**: in casi come quello trattato espandere il dataset può essere un operazione costosa dal punto di vista temporale ma necessaria vista la grande quantità di dati necessari per poter allenare correttamente un modello. La *data augmentation* permette di estendere il dataset, simulando condizioni realistiche a partire da quelle preesistenti.

Overfitting: introdurre modifiche casuali impedisce al modello di imparare a riconoscere le specifiche immagini costringendolo a generalizzare ciò che impara rendendolo anche più resistente alle variazioni dei dati

```
train_generator = train_datagen.flow_from_directory(
        ./train3',
        target_size=(100, 100),
        batch_size=batch_size
        class_mode='categorical',
        shuffle=True)
```

Questa sezione di codice ottiene le immagini dalla cartella desiderata e le utilizza come input per la *data augmentation*. Inoltre viene decisa la modalità con cui si andranno a produrre i risultati (la scelta è tra classificazione binaria o multiclasse). Grazie a *shuffle=True* ci si assicura che il dataset venga rimescolato in modo tale da aumentare la casualità ed evitare possibili bias nel processo di apprendimento. Una sezione di codice analoga viene utilizzata per le immagini di validazione con la differenza che il rimescolamento viene disattivato perché, oltre a risultare inutile durante la procedura di valutazione, si evita che le predizioni vengano associate incorrettamente alle immagini risultando così in risultati non validi nonostante la loro possibile correttezza.

```
history = model.fit(train_generator, epochs=30
                    ,validation_data=validation_generator,
                    callbacks=[stop_callback, checkpoint_callback])
predictions = model.predict(validation_generator)
```

Il metodo .fit si occupa di eseguire il *training* impostando alcuni parametri tra cui: dove recuperare le immagini di *training* e *validation* il numero di epoche e le *callback* utilizzate. Invece il metodo *predict* esegue le predizioni utilizzando le immagini di *validation*.

6.3.3 CNN ausiliari

Come detto gli altri modelli sono molto simili, con l'eccezione di alcuni cambiamenti:

CNN per i 22 slot piccoli: viene ridotto il *dropout* da 0.1 a 0.2, vengono utilizzati i *normalization layer* e viene ridotto il numero delle *epoch* da 30 a

15.

```
layers.BatchNormalization(),
```

CNN per i 2 slot grandi: viene eliminato il *dropout*, non vengono utilizzati i *normalization layer*, il numero di *epoch* rimane invariato a 30.

6.3.4 CNN cestino

Quest'ultimo modello, realizzato con l'intento di essere uno studio di fattibilità, si occupa invece del riconoscimento dello stato di riempimento del cestino per le punte monouso integrato in macchina. Questo modello infatti deve poter rilevare quanto il cestino sia pieno in modo tale da avvisare l'utente di svuotarlo. Dopo dei risultati iniziali non promettenti (0,33 di *accuracy*) alla rimozione dei layer di *dropout* e di *normalization* hanno permesso il raggiungimento di risultati interessanti (0,883 di *accuracy*). Una delle principali difficoltà incontrate è dovuta alla dimensione delle immagini: essendo loro 814×209 pixel e non 100×100 la dimensione dei *batch* è stata ridotta da 30 a 20 e sia il numero di filtri che il numero di *layer* a cui sono stati applicati è stato drasticamente ridotto, in caso contrario la memoria disponibile non sarebbe stata sufficiente. A causa di queste riduzioni il numero di *epoch* è stato aumentato a 40 cosa che, unita alle dimensioni delle immagini, ha reso il training molto lento.

```
model = Sequential([
    layers.Input(shape=(814, 209, 3)),
    layers.Conv2D(16, (2,2), padding='same', activation='relu'),
    layers.MaxPooling2D(padding='same'),
    layers.Conv2D(32, (3,3), padding='same', activation='relu'),
    layers.MaxPooling2D(padding='same'),
    layers.Conv2D(64, (3,3), padding='same', activation='relu'),
    layers.MaxPooling2D(padding='same'),
    layers.Conv2D(128 ,3, padding='same', activation='relu'),
    layers.Flatten(),
```

```
    layers.Dense(128, activation='relu'),
    layers.Dense(3, activation='softmax')
    ])
```

Capitolo 7

Testing

7.1 CNN primari

Questa prima rete neurale è stata allenata su 6000 immagini e validata su 1500 immagini suddivise in 5 classi (*cap, empty, small, medium, large*). Grazie alla grande quantità di immagini acquisite è stata una delle reti che ha portato i risultati migliori come si può vedere in Fig.7.1 e in Fig.7.2. In particolare si può notare come la *validation accuracy* raggiunge il 99,3% mentre la *validation loss* raggiunge lo 0,36%.

Figura 7.1: Grafico dell'*accuracy* del modello al passare delle *epoch*

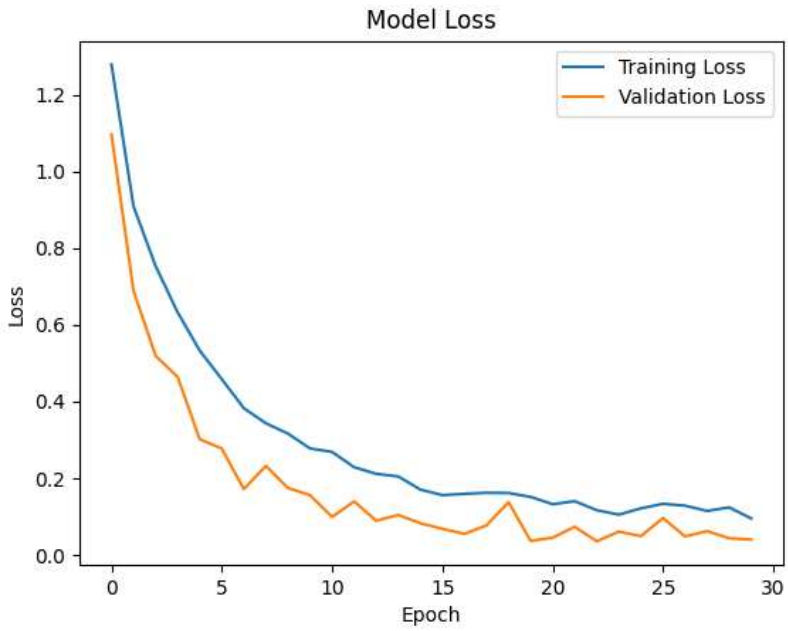

Figura 7.2: Grafico della *loss* del modello al passare delle *epoch*

Questa è la rete neurale che si occupa di dover gestire la più grande varietà di classi e di variazioni delle caratteristiche delle provette anche all'interno della stessa classe. I problemi principali riscontrati sono riassumibili nelle seguenti immagini (Fig.7.3 e Fig.7.4):

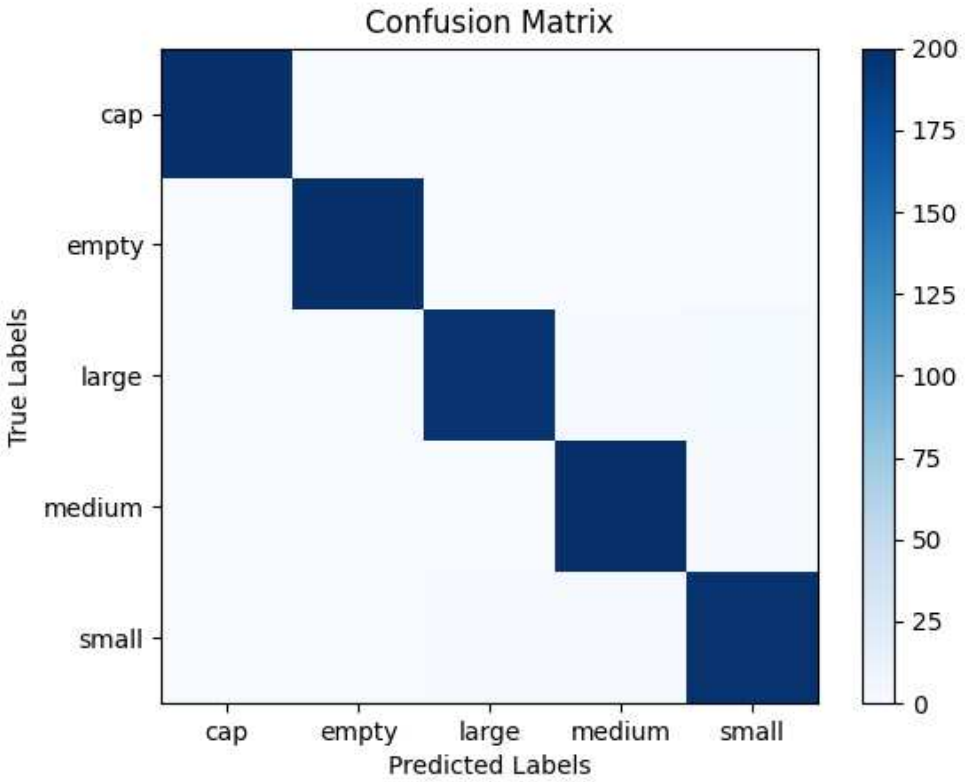

Figura 7.3: *Confusion matrix* della CNN

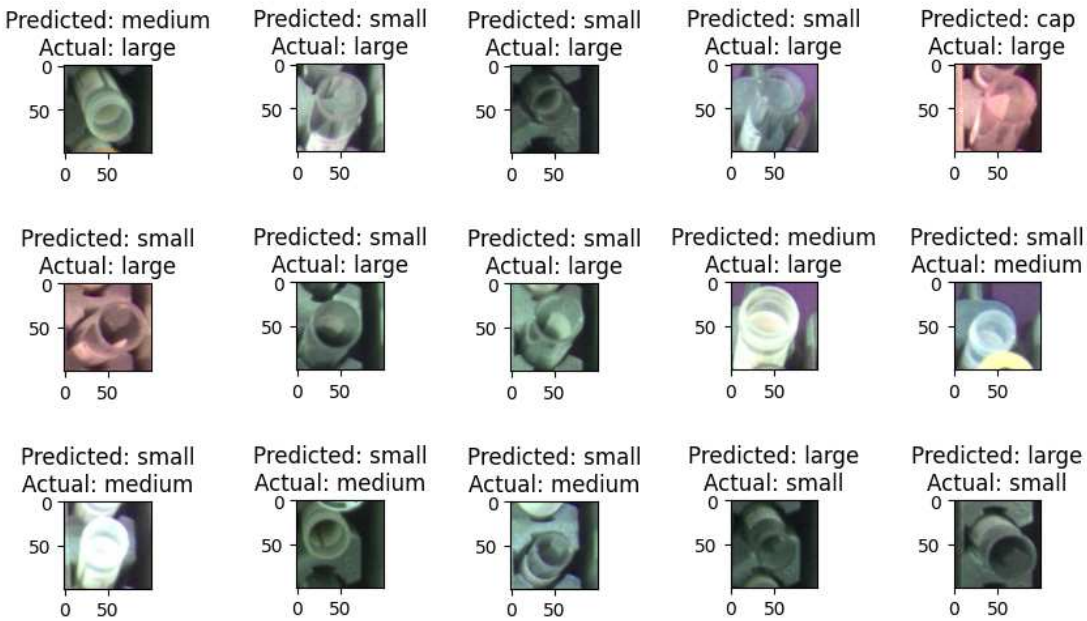

Figura 7.4: Esempi di errori commessi dalla CNN nella classificazione

Come si può notare nella Fig. 7.4 gli errori più presenti sono quelli relativi alle

provette trasparenti, presenti sia nella classe *large* che nella classe *small*, e quelli relativi ai riflessi molto intensi.

7.2 CNN ausiliari (provette piccole)

Questa rete neurale è stata allenata su 4500 immagini e validata su 600 immagini suddivise in 3 classi (cap, empty, no_cap). Questa rete neurale, grazie alla grande omogeneità delle provette, delle quali non cambia mai la dimensione ma solo il colore del tappo come si vede in Fig. 7.5 ha raggiunto risultati eccellenti (100% di *accuracy*, Fig.7.6 e una *loss* trascurabile Fig.7.7) venendo anche impiegata sul campo.

Figura 7.5: Esempi delle diverse provette

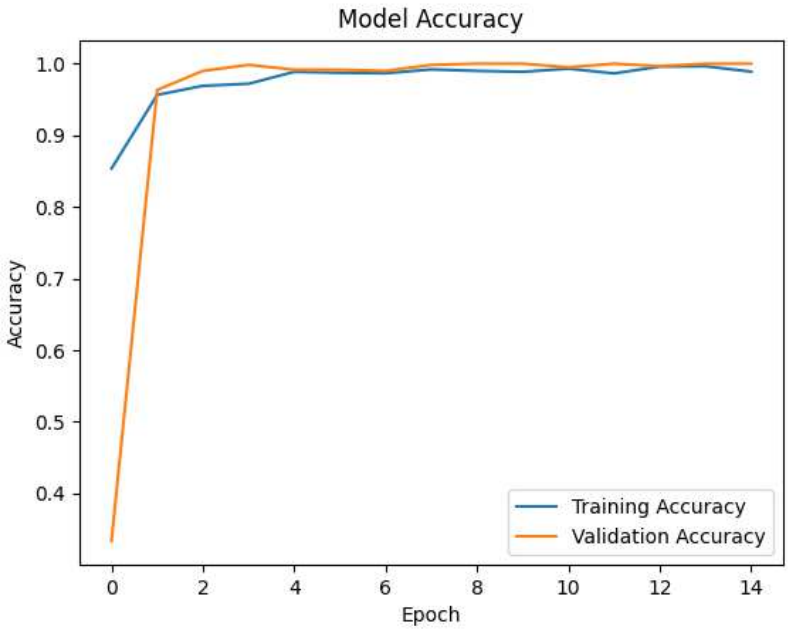

Figura 7.6: Grafico della *accuracy* del modello al passare delle *epoch*

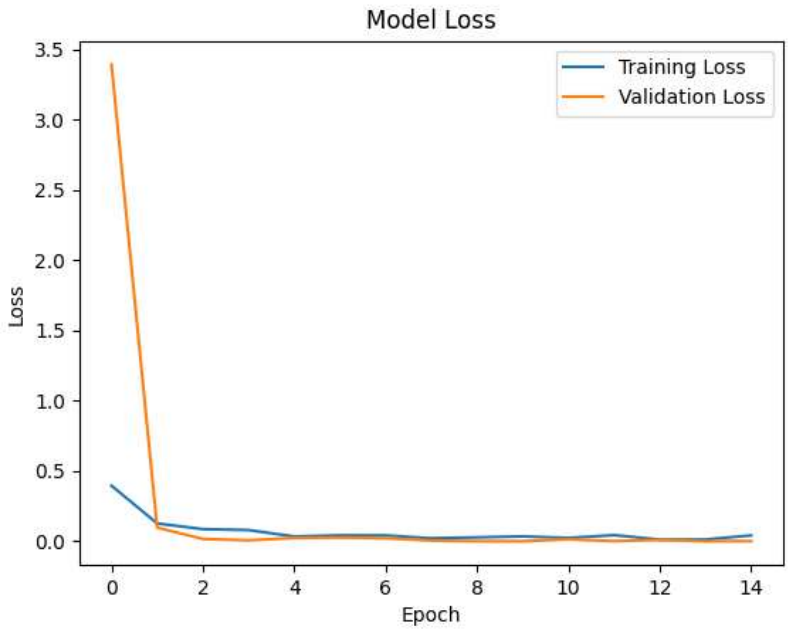

Figura 7.7: Grafico della *loss* del modello al passare delle *epoch*

7.3 CNN ausiliari (provette grandi)

Questa rete neurale è stata allenata su 1200 immagini e validata su 210 immagini suddivise in 3 classi (*cap, large, no_cap*). Dato il campione ridotto, dovuto alle poche posizioni disponibili per le provette e alle poche tipologie di provette utilizzabili le quali erano molto diverse fra loro, questa rete ha prodotto risultati leggermente inferiori alla prima CNN proposta, in particolare l'*accuracy* è del 98,1% mentre la *loss* è pari al 10,7%. Altro fattore da notare sono i grafici che, come si vede in Fig. 7.8 e Fig. 7.9 sono molto più irregolari.

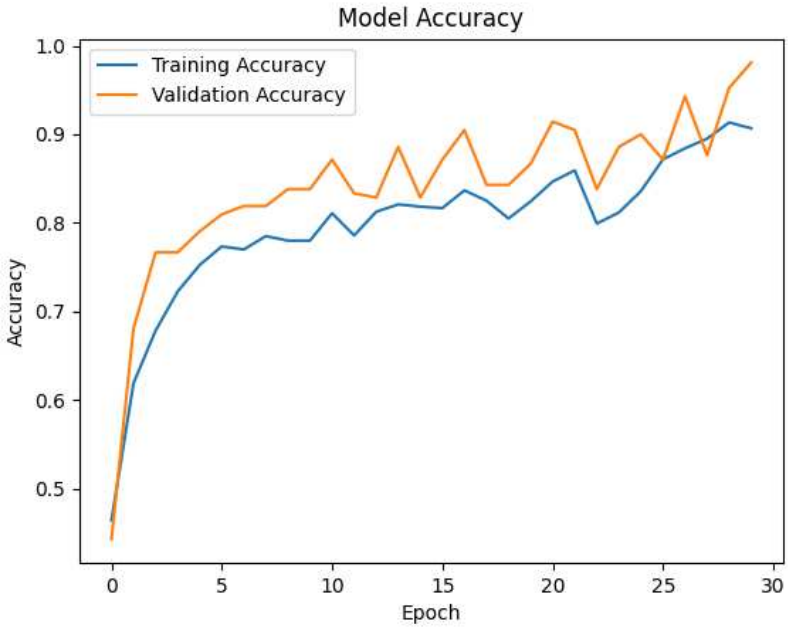

Figura 7.8: Grafico della *loss* del modello al passare delle *epoch*

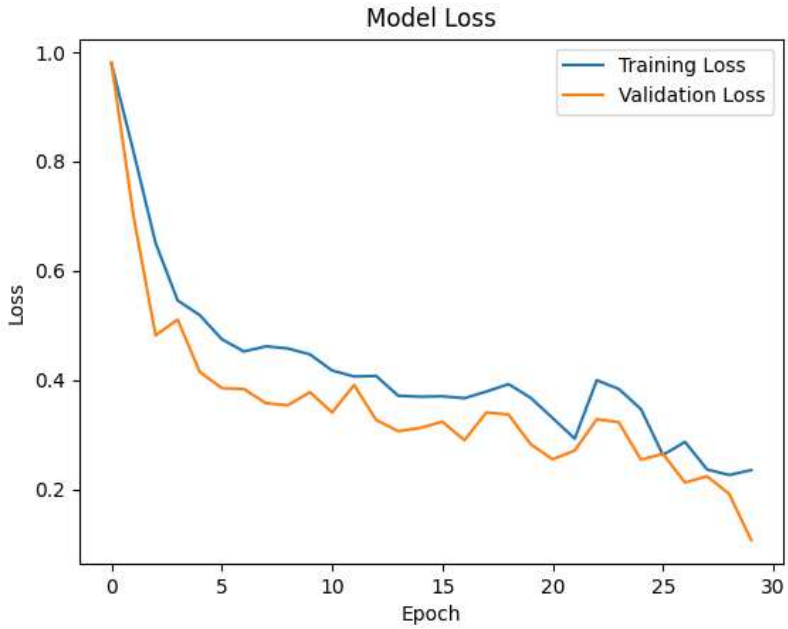

Figura 7.9: Grafico della *loss* del modello al passare delle *epoch*

Gli errori principali commessi da questa rete sono visibili nelle immagini Fig. 7.10 e Fig. 7.11, dove si nota come siano le condizioni di luce estreme a mettere in difficoltà la rete.

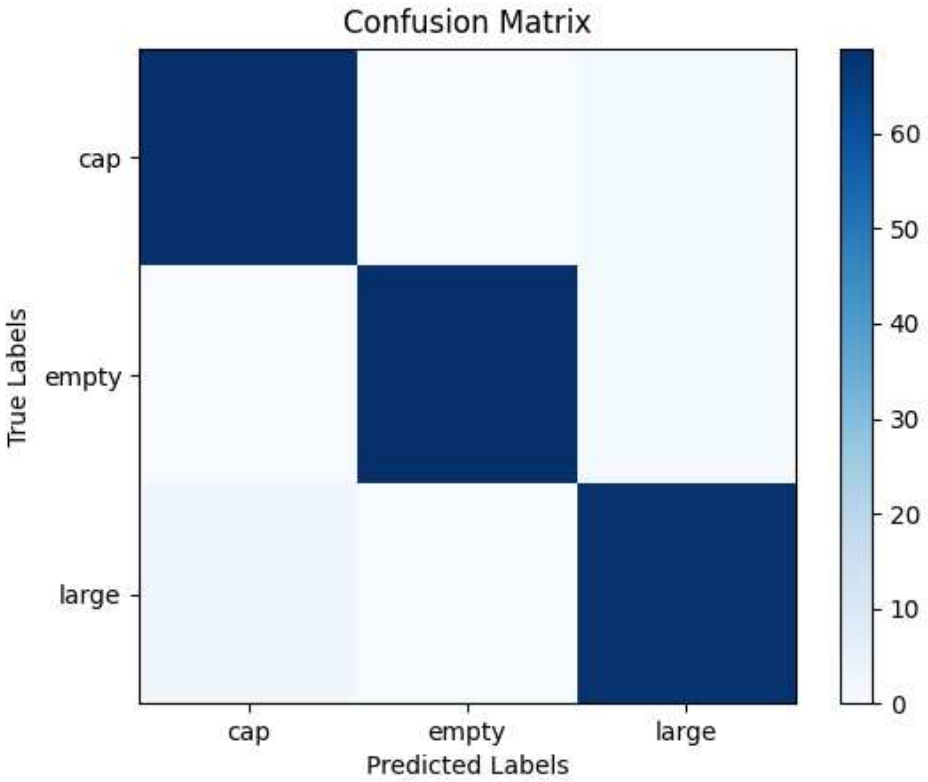

Figura 7.10: *Confusion matrix* della CNN

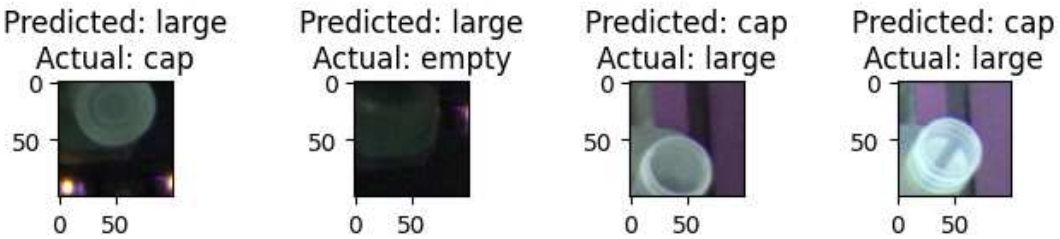

Figura 7.11: Esempi di errori commessi dalla CNN nella classificazione

7.4 CNN cestino

Questa rete neurale è stata allenata su 450 immagini e validata su 120 immagini suddivise in 3 classi (*empty, not_empty, full*). Come detto in precedenza questa rete neurale è stata sviluppata allo scopo di verificare se fosse possibile identificare lo stato di riempimento del cestino. Inizialmente ciò non sembrava essere possibile anche a causa di difficoltà riguardanti la creazione del dataset, come ad esempio la mancanza dei sacchetti (in seguito reperiti) della tipologia giusta oppure la forma del cestino che forniva solo una piccola apertura per poterne osservare lo stato interno, tuttavia dopo vari tentativi è stato determinato che fosse effettivamente possibile raggiungere lo scopo ottenendo un 88,3% di *validation accuracy* e una *loss* pari al 45,31% come si può vedere nelle figure Fig. 7.12 e Fig. 7.13

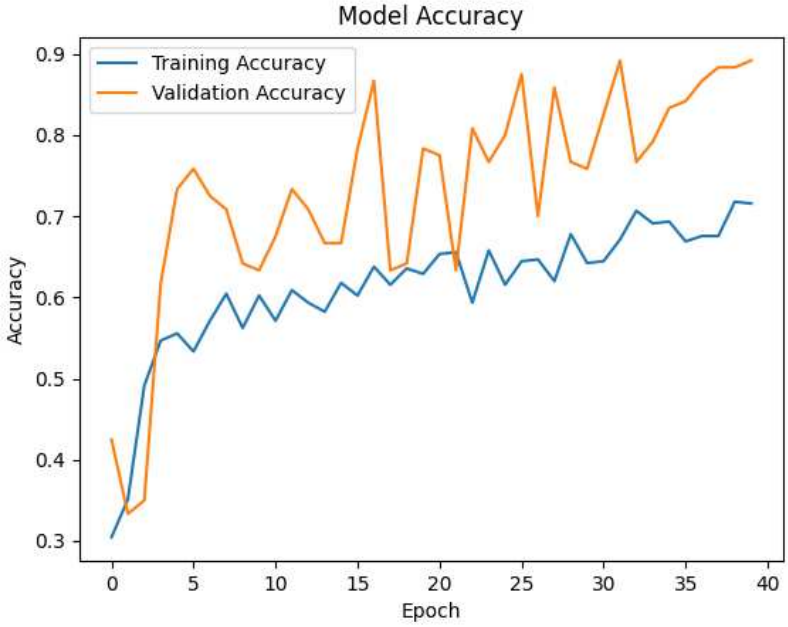

Figura 7.12: Grafico della *accuracy* del modello al passare delle *epoch*

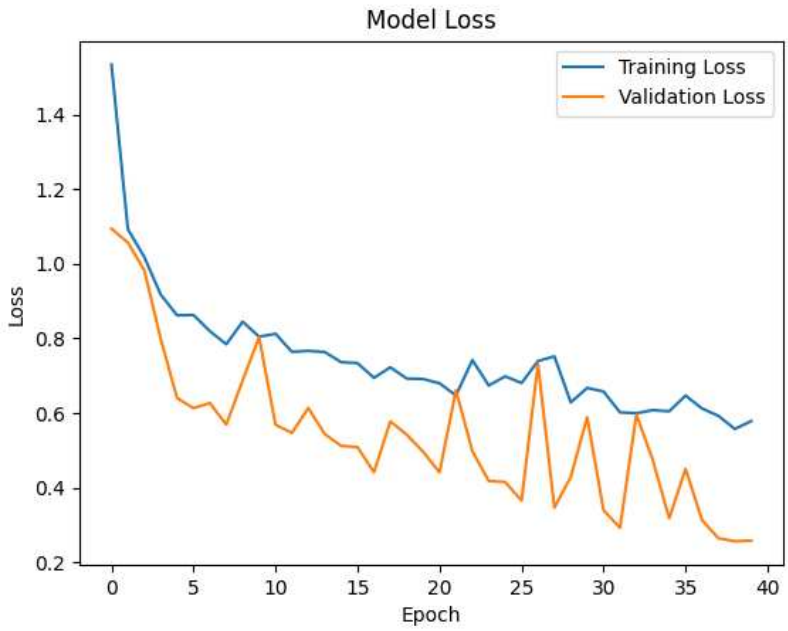

Figura 7.13: Grafico della *loss* del modello al passare delle *epoch*

Le difficoltà di classificazione maggiori incontrate da questo modello sono quelle relative all'identificazione della classe *not_empty*: infatti la totalità degli errori consiste nell'identificazione di questa classe come una delle altre due oppure viceversa senza mai confondere tra loro *empty* e *full* come si vede in Fig. 7.14 e Fig. 7.15. Questo è dovuto a due fattori: il primo è la similarità tra le foto, infatti aggiungendo un puntale alla volta le differenze sono minimali e inoltre difficili da notare a causa del fatto che la colorazione dei puntali è nera come quella del cestino stesso. Il secondo fattore è invece l'arbitrarietà nella distinzione delle tre classi: è infatti semplice capire quando una foto rappresenta un cestino completamente vuoto o molto pieno ma è più difficile determinare l'esatta quantità di puntali che distinguono un cestino *not_empty* da uno *full* o identificare il singolo puntale presente in una foto.

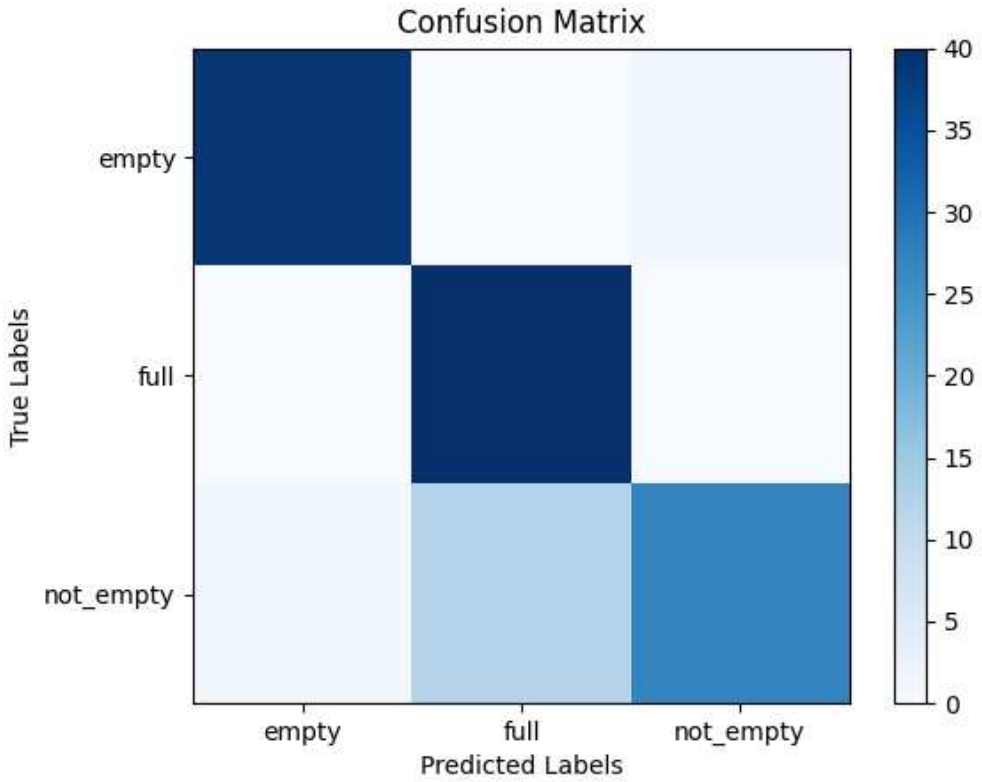

Figura 7.14: *Confusion matrix* della CNN

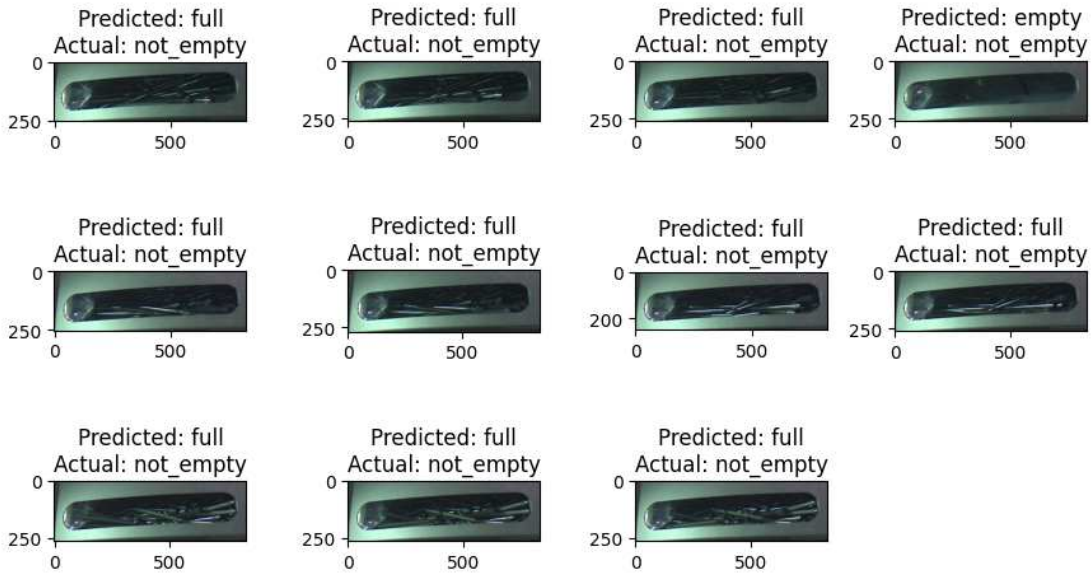

Figura 7.15: Esempi di errori commessi dalla CNN nella classificazione

7.5 Osservazioni

Come si può notare dai grafici Fig. 7.1, 7.2,7.6,7.7,7.8,7.9,7.12 e 7.13 i valori delle metriche (sia per quanto riguarda l'*accuracy* che per quanto riguarda la *loss*) ottenuti durante la validazione sono sempre migliori di quanto non lo siano i corrispettivi valori ottenuti durante il training e ciò vuol dire che i modelli sono in grado di riconoscere meglio campioni mai visti prima rispetto a valori già visti il che è inusuale. Questo normalmente è dovuto a errori di progettazione come ad esempio il *data bleeding*, fenomeno per il quale il modello sfrutta informazioni esterne alle immagini (come ad esempio prossimità spaziale o temporale tra esse) per effettuare le previsioni, cosa che può essere un problema in quanto, solitamente, le immagini si ottengono a "blocchi" connessi secondo un qualche tipo di logica, come ad esempio "tutte le provette con il tappo". Tuttavia questo non è possibile in questo caso in quanto, come spiegato in precedenza, i *crop* utilizzati per il *training* e per la validazione sono generati tutti contemporaneamente da uno *script* e vengono poi selezionati casualmente da un secondo *script* per creare il *training* e il *validation set*. Ulteriore sicurezza da questo punto di vista è l'introduzione di un rimescolamento del *training set* dopo ogni *epoch*. Tenendo conto di tutto ciò la causa più probabile di questa peculiarità è l'utilizzo del *data agumentation*: infatti esso è stato utilizzato solo per espandere il *training set* senza venir applicato al *validation set*, rendendo quindi più complesse e variabili le immagini su cui è stato allenato il modello rispetto a quelle su cui è stato testato.

Capitolo 8

Conclusioni

Il presente lavoro di tesi aveva l'obiettivo di automatizzare GENEQUALITY® TWENTYFIFTY (GQ 2050) tramite l'utilizzo dell'intelligenza artificiale, riducendo gli errori umani e aumentando la velocità con cui le operazioni vengono eseguite. Come discusso in precedenza l'obiettivo è stato raggiunto in alcuni casi mentre in altri ne è stata dimostrata la fattibilità (nel caso del cestino). Nonostante i buoni risultati raggiunti si possono trattare alcuni possibili miglioramenti come ad esempio l'aggiunta di una classe riflesso nelle reti neurali predisposte al riconoscimento delle provette: se ad esempio la classificazione è incerta (ad esempio con una *confidence*, ovvero la probabilità che la predizione sia corretta, inferiore al 75%) si potrebbe segnalare all'utente di controllare manualmente una determinata provetta. Per quanto riguarda la CNN del cestino alcuni miglioramenti possibili sono aumentare le dimensioni del *dataset* (cosa che richiederebbe anche un aumento della potenza dell'*hardware* disponibile in modo da poter rendere più complessa la rete neurale e aumentare la velocità di *training*) oppure cambiare il colore del sacchetto utilizzato per il raccoglimento dei puntali. Quest'ultima soluzione tuttavia non è praticabile in quanto il sacchetto è stato scelto da AB analitica e non è modificabile. Il sacchetto inoltre copre il contenitore metallico che costituisce il cestino rendendo irrilevante la colorazione di quest'ultimo.

Per quanto riguarda lo stage invece sono stati raggiunti alcuni obiettivi: innanzitutto ho raggiunto una maggiore conoscenza relativa alla struttura e al funzionamento delle CNN e in secondo luogo ho potuto vedere applicate sul campo alcune meto-

dologie dello sviluppo *software* (come ad esempio gli *stand up meeting*) applicate in un contesto lavorativo reale il quale è stato fin da subito un ambiente positivo e inclusivo.

Bibliografia

Libri

Carbonell, Jaime. *Machine learning*. Tom M. Mitchell, Jaime Carbonell, Ryszard Michalski, 1997 (cit. a p. 3).

Fioresi, Rita e F. Zanchetta. *Deep Learning and Geometric Deep Learning: an introduction for mathematicians and physicists*. Mag. 2023. DOI: 10.48550/arXiv.2305.05601 (cit. a p. 24).

Siti

AdaGrad. URL: https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827 (cit. a p. 20).

AdaGrad. URL: https://towardsdatascience.com/understanding-deep-learning-optimizers-momentum-adagrad-rmsprop-adam-e311e377e9c2 (cit. a p. 20).

Adam. URL: https://www.geeksforgeeks.org/adam-optimizer-in-tensorflow/ (cit. a p. 22).

Adam. URL: https://www.sciencedirect.com/science/article/abs/pii/S0020025522006260 (cit. a p. 22).

Adam. URL: https://it.wikipedia.org/wiki/Keras (cit. a p. 22).

Balanced accuracy. URL: https://www.statology.org/balanced-accuracy/ (cit. a p. 9).

Dying relu. URL: https://www.geeksforgeeks.org/what-is-the-dying-relu-problem-in-neural-networks/ (cit. a p. 14).

F1-score, precision, recall. URL: https://medium.com/@nirajan.acharya777/understanding-precision-recall-f1-score-and-support-in-machine-learning-evaluation-7ec935e8512e (cit. a p. 8).

Keras. URL: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/.

Leaky relu. URL: https://www.v7labs.com/blog/neural-networks-activation-functions/ (cit. a p. 15).

opt. URL: https://builtin.com/data-science/gradient-descent (cit. a p. 17).

Precision-recall tradeoff. URL: https://medium.com/analytics-vidhya/precision-recall-tradeoff-79e892d43134 (cit. a p. 8).

RMSProp. URL: https://www.geeksforgeeks.org/rmsprop-optimizer-in-deep-learning/ (cit. a p. 21).

RMSProp. URL: https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a (cit. a p. 21).

RProp. URL: https://florian.github.io/rprop/ (cit. a p. 18).

RProp. URL: https://en.wikipedia.org/wiki/Rprop (cit. a p. 18).

Sensitivity and specificity. URL: https://towardsdatascience.com/evaluating-categorical-models-ii-sensitivity-and-specificity-e181e573cff8.

Tensorflow. URL: https://it.wikipedia.org/wiki/TensorFlow (cit. a p. 37).