Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

UN'APPLICAZIONE iOS PER IL RICONOSCIMENTO DI MEDICINALI MEDIANTE RETI NEURALI

Relatore:
Chiar.mo Prof.
LUCIANO BONONI

Presentata da:
MATTEO DEL VECCHIO

Correlatore:
Dott. LUCA BEDOGNI

Sessione II
Anno Accademico 2016/2017

A mia sorella Martina
A mia madre Maria Rosaria
A mio padre Giovanni

Introduzione

Nel corso degli ultimi anni abbiamo assistito all'affermazione progressiva di una branca molto vasta dell'informatica chiamata Machine Learning, avente come obiettivo l'elaborazione di meccanismi tali da permettere alle macchine di apprendere, senza che esse siano esplicitamente programmate per fare ciò. Nello specifico, la loro capacità di imparare si basa essenzialmente sulla creazione di un'esperienza a partire da migliaia di esempi, da cui successivamente si ottengono previsioni o decisioni.
Sebbene l'apprendimento automatico sia correlato a molti ambiti, anche esterni all'informatica, esso viene principalmente impiegato in quei settori nei quali progettare un algoritmo o una soluzione è perlopiù impraticabile. Tale è il caso della Visione Artificiale, dove è spesso richiesto di identificare gli oggetti contenuti in un'immagine o, in alternativa, ricostruire un modello 3D partendo da dati in 2D.
Nella fattispecie, Machine Learning è un termine coniato nel 1959 da A. L. Samuel [1] e molti dei principali algoritmi sono stati teorizzati negli anni successivi. In realtà, si è dovuto attendere la disponibilità della capacità computazionale necessaria per poterli sfruttare al meglio, insieme ad una grande mole di dati sotto forma di dataset. Tra questi algoritmi ci sono le Reti Neurali Artificiali, ed in particolare le Convolutional Neural Network (anche chiamate CNN o ConvNet), che sono alla base del lavoro svolto in questa tesi, poichè dimostratesi particolarmente performanti nel riconoscimento e nella classificazione di immagini digitali, già dalle prime versioni proposte, come la LeNet-5 di LeCun et al [2]. Un importante avanzamento va attribuito ad

Alex Krizhevsky [3] che ha avuto l'intuizione secondo cui anche parametri come la profondità, possono notevolmente influenzare le capacità di una rete neurale convoluzionale. Ciò gli ha permesso di vincere l'ImageNet LSVRC-2010 [4] con la sua rete neurale.
Di ImageNet e di alcune reti considerate allo stato dell'arte se ne accennerà nel Capitolo 1, insieme ad una discussione sulle caratteristiche delle ConvNet. Nel Capitolo 2 verrà invece descritta PillRecogNet, ossia la CNN creata al fine di riconoscere i medicinali partendo dalle foto degli stessi, ed i relativi strumenti utilizzati per il training. Infine, nel Capitolo 3, ci si concentrerà in ambito iOS, esponendo la realizzazione di PillRecogNet in un'applicazione vera e propria, con particolare riguardo verso le possibili scelte implementative e ai requisiti correlati.

Indice

Introduzione **i**

1 Convolutional Neural Networks **1**

1.1 Concetti Basilari 2

1.1.1 Neurone 2

1.1.2 Funzione di Attivazione 3

1.2 Multilayer Perceptron 4

1.3 Dimensione di una Rete Neurale 6

1.4 Caratteristiche delle ConvNet 6

1.4.1 Convolutional Layer 7

1.4.2 Pooling Layer 8

1.4.3 Fully Connected Layer 9

1.5 Training 10

1.6 Stato dell'Arte 11

1.6.1 ImageNet 11

1.6.2 GoogLeNet / Inception V1 12

1.6.3 Inception V3 12

1.6.4 VGGNet 13

2 PillRecogNet **15**

2.1 Creazione del Dataset 15

2.2 Strumenti di Sviluppo 17

2.3 Transfer Learning 19

2.3.1 Preprocessing 22

2.3.2 Feature Extraction 23
2.3.3 Fine Tuning 23
2.4 Statistiche e Risultati 24

3 Applicazione iOS 29
3.1 ConvNet in Ambito iOS 29
3.1.1 Basic Neural Network Subroutines (BNNS) 30
3.1.2 Metal Performance Shaders (MPS) 31
3.1.3 CoreML e Vision 32
3.2 Scelte Implementative 33
3.3 PillRecogNet App 36

Conclusioni 41

Sviluppi Futuri 43

A PillRecogNet in Dettaglio 45
A.1 Struttura Generale 45
A.2 Livelli Finali Personalizzati 47

B Implementazione 49
B.1 Python 49
B.1.1 Feature Extraction 49
B.1.2 Fine Tuning 51
B.1.3 Valutazione della Rete Neurale 54
B.1.4 Esportazione Parametri 56
B.1.5 Codice Condiviso 57
B.2 Swift e Metal 59
B.2.1 PillRecogNet 59
B.2.2 PillLabelManager 68
B.2.3 Preprocessing Kernel 69

Bibliografia 71

Elenco delle figure

1.1 Rappresentazione di una rete neurale convoluzionale 1
1.2 Modello matematico di un neurone 2
1.3 Funzione *sigmoid* 3
1.4 Funzione *tanh* 3
1.5 Funzione *ReLU* 4
1.6 Convergenza *ReLU* 4
1.7 Esempio di Multilayer Perceptron 4
1.8 Esempio di applicazione di un filtro 7
1.9 Esempio convoluzione 9
1.10 Esempio *pooling* 10
1.11 Modulo Inception 12

2.1 Esempio dataset di *training* 16
2.2 Esempio manipolazione delle immagini 17
2.3 Struttura VGG16 18
2.4 Visualizzazione di alcune *feature* 19
2.5 *Transfer learning* 20
2.6 Esempio applicazione *dropout* 24
2.7 *Learning rate* dinamico 26
2.8 Statistiche relative al *training* 26
2.9 Esempi di classificazione corretta 27
2.10 Esempi di classificazione errata 28

3.1 Schema utilizzo CoreML 32

3.2 Schermate principali dell'applicazione iOS 36
3.3 Esperienza d'uso dell'applicazione iOS 38
3.4 Ulteriori schermate di dettaglio 39

Elenco dei listati

2.1 Creazione VGG16 con parametri 21
2.2 Sottrazione dei valori RGB medi 22
2.3 Definizione rete neurale aggiuntiva 24

3.1 Esempio livello convoluzionale tramite BNNS 30
3.2 Esempio livello convoluzionale tramite MPS 31
3.3 Esempio definizione ed *encoding* tramite MPS 33

B.1 Python - Estrazione delle *feature* 49
B.2 Python - *Fine tuning* della rete neurale 51
B.3 Python - Valutazione dell'accuratezza sul dataset di test . . . 54
B.4 Python - Esportazione dei parametri per MPS 56
B.5 Python - Impostazioni per il *training* 57
B.6 Python - Funzioni condivise 58
B.7 Swift - Implementazione di PillRecogNet con MPS 59
B.8 Swift - Gestore delle classi riconosciute 68
B.9 Metal Shading Language - Sottrazione dei valori RGB medi . . 69

Elenco delle tabelle

1.1 Confronto reti neurali 13

2.1 Risultati accuratezza 27

Capitolo 1

Convolutional Neural Networks

Le reti neurali convoluzionali rappresentano una classe di reti neurali definite *deep* e *feed-forward*, ovvero formate da numerosi livelli di profondità in cui le connessioni avvengono in una sola direzione, non permettendo la formazione di cicli. La loro applicazione risulta essere particolarmente performante nell'analisi e nel riconoscimento di immagini e video.
Verranno illustrate qui di seguito alcune nozioni necessarie alla comprensione della struttura di una ConvNet.

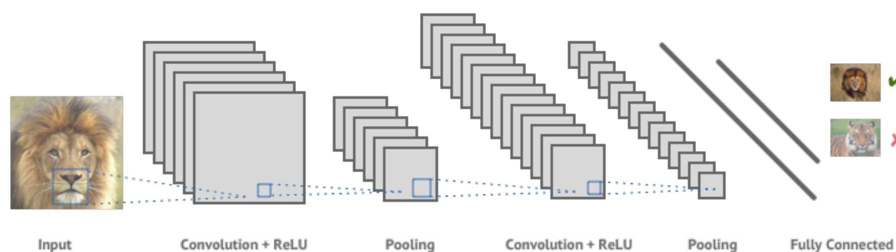


Figura 1.1: Rappresentazione di una rete neurale convoluzionale[1]

[1]Immagine proveniente da `https://shafeentejani.github.io/2016-12-20/convolutional-neural-nets/`

1.1 Concetti Basilari

Come si evince dal nome, le reti neurali sono ispirate dai processi biologici e, nel caso di quelle convoluzionali, dalla corteccia visiva dove si hanno dei neuroni che rispondono agli stimoli esterni della vista. In modo analogo, si avranno concetti tali da simularne il comportamento.

1.1.1 Neurone

Il neurone può essere considerato l'elemento fondamentale delle reti neurali, in quanto alla base dei livelli di cui esse sono composte.
Il principio di funzionamento consiste nel ricevere un segnale in input (x_0), elaborarlo e inviarlo in output ad altri neuroni. La connessione utilizzata dal segnale per arrivare ad un neurone ne modifica l'intensità, in quanto a ognuna è associato un peso (w_0) che viene moltiplicato ($x_0 w_0$) con il segnale stesso. Una volta ricevuti al completo, l'elaborazione del neurone risulta essere la somma degli input e di un eventuale *bias* (b), un parametro associato al neurone stesso, così come accade per le connessioni. L'idea di fondo risiede nell'assegnare e modificare i pesi delle connessioni e dei neuroni in modo tale da ottenere uno specifico output: così facendo si modella la capacità di apprendere delle reti neurali. In termini biologici, le connessioni in input rappresentano i *dendriti*, quelle in output gli *assoni* mentre i pesi su di esse, le *sinapsi*.

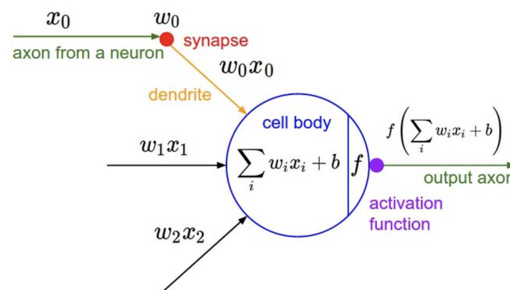


Figura 1.2: Modello matematico di un neurone[2]

1.1.2 Funzione di Attivazione

La funzione di attivazione (f) di un neurone rappresenta un ulteriore passo di elaborazione, applicata prima che il risultato sia inviato ai neuroni successivi; si tratta di funzioni non lineari (per questo motivo chiamate anche *non-linearity*) che applicano una precisa manipolazione matematica ai dati in input. Storicamente, le più utilizzate sono la *sigmoid* (1.1) e la *tanh* (1.2) che hanno la particolarità di rapportare i valori in input ad intervalli precisi cioè, rispettivamente, $[0, 1]$ e $[-1, 1]$.

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \quad (1.1) \qquad tanh(x) = \frac{2}{1 + e^{-2x}} - 1 \quad (1.2)$$

Recentemente, tali funzioni sono state spesso sostituite dall'utilizzo della Rectified Linear Unit (1.3), o ReLU, grazie al lavoro svolto da Krizhevsky at al. [3] e Glorot et al. [5], dimostrando che le reti neurali basate sulla suddetta funzione di attivazione comportano una più veloce convergenza e, di conseguenza, una maggiore velocità di allenamento su dataset complessi. In particolare, essa sarà la funzione utilizzata in PillRecogNet ed è semplicemente definita come:

$$ReLU(x) = max(0, x) \quad (1.3)$$

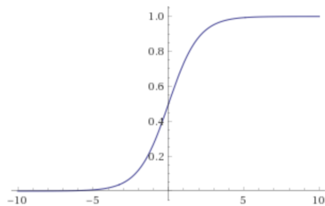


Figura 1.3: Funzione *sigmoid*[3]

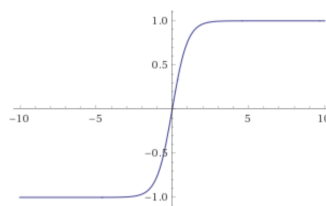


Figura 1.4: Funzione *tanh*[3]

[2]Immagine proveniente da `http://cs231n.github.io/neural-networks-1/`
[3]Immagine proveniente da `http://www.wolframalpha.com`

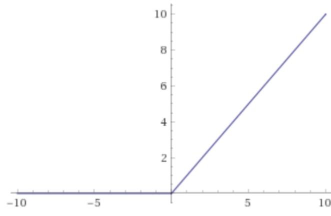


Figura 1.5: Funzione *ReLU*[3]

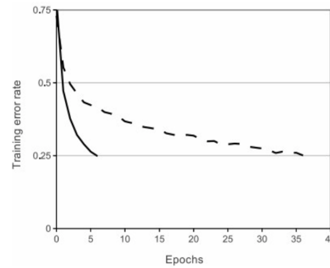


Figura 1.6: Convergenza *ReLU* [3]

1.2 Multilayer Perceptron

L'idea delle reti neurali convoluzionali è basata sul Multilayer Perceptron, un più semplice tipo di rete neurale *feed-forward* composta da almeno due livelli di neuroni (per convenzione, il livello di input non viene contato). In generale, si hanno un livello di input, un livello di output ed uno o più *hidden layer*, ovvero quei livelli che applicano le trasformazioni necessarie allo scopo di ottenere l'output desiderato. Utilizzando funzioni di attivazione non lineari come quelle precedentemente trattate, una rete neurale di questo tipo permette di approssimare una funzione non lineare che possa separare i dati secondo la loro reale classe. Per questo motivo, l'aumento del numero degli *hidden layer* consente di rappresentare funzioni sempre più complesse aumentando, al contempo, la dimensione della rete stessa.

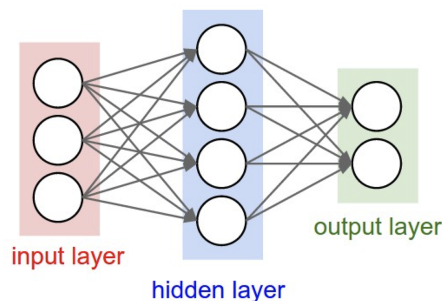


Figura 1.7: Esempio di Multilayer Perceptron da due livelli[4]

[4]Immagine proveniente da `http://cs231n.github.io/neural-networks-1/`

Dalla Figura 1.7 è possibile distinguere alcune peculiarità: in tutti gli *hidden layer* ogni neurone ha un numero di connessioni in ingresso pari al numero di quelli del livello precedente; allo stesso modo, tante connessioni in uscita quanti sono i neuroni del livello successivo. Inoltre, neuroni allo stesso livello non hanno connessioni tra loro. Un *layer* avente una tale configurazione è detto *fully connected layer*. Un discorso simile si applica anche ai livelli di input ed output, con la differenza che non ci sono connessioni in ingresso nel primo caso, e in uscita nel secondo. In più, quest'ultimo non è quasi mai dotato della funzione di attivazione perché è interessante il valore dei neuroni in sé, che rappresentino una distribuzione di probabilità (classificazione) o un valore arbitrario (regressione).
Questa particolare disposizione permette di generalizzare l'elaborazione svolta da un singolo neurone, dando lungo ad un calcolo matriciale: difatti, si indicano i pesi attraverso una matrice $\boldsymbol{W}$ di dimensione $n \times m$ (dove n indica il numero di neuroni al livello corrente ed m al livello precedente), i valori dei neuroni al livello precedente con un vettore $\underline{a}$ ed i *bias* del livello corrente con un vettore $\underline{b}$. Le connessioni tra il livello di input e l'*hidden layer* nella rete in Figura 1.7 danno luogo ad una matrice $\boldsymbol{W}$ 4×3, al vettori $\underline{a}$ 3×1 e al vettore $\underline{b}$ 4×1. In generale:

$$\underline{a}^{(k)} = f(\begin{pmatrix} w_{1,1} & \dots & w_{1,m} \\ \vdots & \ddots & \vdots \\ w_{n,1} & \dots & w_{n,m} \end{pmatrix} \begin{pmatrix} a_1^{(k-1)} \\ \vdots \\ a_m^{(k-1)} \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}) \tag{1.4}$$

dove f indica la funzione di attivazione, l'elemento $a_i^{(k-1)}$ rappresenta il valore dell'i-esimo neurone al livello $k-1$ ed $\underline{a}^{(k)}$ indica il vettore che descrive i valori di tutti i neuroni al livello k. Più semplicemente, i valori del k-esimo livello sono dati da:

$$\underline{a}^{(k)} = f(\boldsymbol{W}\underline{a}^{(k-1)} + \underline{b}) \tag{1.5}$$

1.3 Dimensione di una Rete Neurale

Ci sono due modi principali per poter calcolare la dimensione di una rete neurale: considerandone il numero di neuroni oppure il numero di pesi e *bias*, dove quest'ultimo è più comunemente utilizzato. Nel primo caso è fondamentale contare il numero di neuroni presenti in tutti i livelli, ad esclusione di quello di input mentre, nel secondo caso, si considerano tutte le connessioni per ottenere il numero dei pesi e i neuroni (esclusi quelli del livello di input) per ottenere i *bias*.
Osservando la Figura 1.7, si hanno:

- $4 + 2 = 6$ neuroni
- $3 \times 4 + 4 \times 2 = 20$ pesi, $4 + 2 = 6$ *bias*, per un totale di 26 parametri

Come termine di paragone, lo stato dell'arte vanta reti neurali aventi centinaia di milioni di parametri, dovuti principalmente alla presenza di decine di livelli. Alcuni esempi verranno mostrati nel corso della trattazione, di cui uno alla base del lavoro svolto per ottenere PillRecogNet.

1.4 Caratteristiche delle ConvNet

Sebbene le CNN abbiano molto in comune con le reti neurali più generiche, ci sono alcune caratteristiche che le differenziano e che le rendono particolarmente efficienti per applicazioni quali il riconoscimento di immagini. Effettivamente, sono state esplicitamente concepite per elaborarle, consentendo quindi la gestione di alcune loro caratteristiche direttamente della struttura della rete stessa.
Tipicamente, un computer "vede" un'immagine come una matrice di valori numerici compresi tra 0 e 255. A tal proposito, è inoltre molto importante il concetto di *canale*: un'immagine in bianco e nero avrà solo il canale della luce mentre, una foto a colori RGB avrà un canale per ogni colore (rosso R, verde G e blu B), pertanto si otterrà una matrice di valori per ognuno dei

canali che compongono l'immagine.
Si introduce quindi l'idea di *profondità*: ogni livello di una rete neurale convoluzionale ha i propri neuroni disposti secondo le tre dimensioni (*larghezza* $\times$ *altezza* $\times$ *profondità*), permettendo di gestire tutti i canali contemporaneamente. In aggiunta, vengono utilizzati tre principali tipi di livelli.

1.4.1 Convolutional Layer

Un livello convoluzionale applica l'operazione matematica della *convoluzione* ai dati che riceve ed essa rappresenta il principio di funzionamento delle ConvNet. Nell'ambito delle reti neurali, si definisce *feature* una qualsiasi informazione rilevante estratta dai dati, utile al calcolo del risultato atteso. La convoluzione, dunque, si occupa di estrarre tali *feature* da ogni immagine in input. Ciò avviene "muovendo" sull'immagine una matrice di dimensione minore, chiamata *kernel*, o *filtro*; essa può essere immaginata come una piccola lampada che passa sull'immagine e ne illumina solo alcune sezioni per volta. Il risultato di tale operazione prende il nome di *feature map*.

Figura 1.8: Esempio di applicazione di un filtro per l'individuazione dei bordi[5]

Una *feature map* dipende tendenzialmente da tre fattori:

1. **Profondità**: indica il numero di *kernel* che devono essere contemporaneamente applicati alla stessa area in esame, le cui *feature map* saranno sovrapposte disponendosi lungo una terza dimensione;

[5] Immagine proveniente da `http://aishack.in/tutorials/image-convolution-examples/`

2. ***Stride***: indica il numero di pixel che devono essere saltati quando il filtro si sposta sull'immagine, riducendone larghezza ed altezza in output;

3. ***Zero-padding***: indica la possibilità di aggiungere un "cuscinetto" di zeri ai bordi dell'input per consentire una corretta applicazione dei *kernel* (caratteristica particolarmente utile in quanto dà modo di controllare, più o meno arbitrariamente, la dimensione del volume in output).

Complessivamente, dato un input di dimensione $\boldsymbol{L_1} \times \boldsymbol{A_1} \times \boldsymbol{P_1}$, un numero di *kernel* $\boldsymbol{K}$, la larghezza $\boldsymbol{D}$ dei *kernel*, uno *stride* $\boldsymbol{S}$ ed un *padding* $\boldsymbol{P}$, un livello convoluzionale calcola un volume in output avente dimensione $\boldsymbol{L_2} \times \boldsymbol{A_2} \times \boldsymbol{P_2}$, con $\boldsymbol{P_2}$ pari a $\boldsymbol{K}$ ed $\boldsymbol{L_2}$ dato dall'equazione 1.6 (in modo analogo anche $\boldsymbol{A_2}$).

$$L_2 = \frac{L_1 - D + 2P}{S} + 1 \tag{1.6}$$

In più, ogni livello avrà un numero di pesi pari a $\boldsymbol{K \cdot D \cdot D \cdot P_1}$ e $\boldsymbol{K}$ *bias*. L'aspetto interessante delle reti neurali convoluzionali consiste nel fatto che tali filtri non sono scelti a priori, bensì si lascia alla rete stessa l'abilità di individuare i valori che permettono una migliore estrazione delle *feature*. Per di più, un neurone è collegato solo con quelli che rientrano nell'area del filtro utilizzato.

1.4.2 Pooling Layer

Un livello di *pooling* viene solitamente utilizzato in seguito ad alcuni livelli convoluzionali per ottenere un duplice beneficio: effettua un *downsampling* dell'input, riducendone la dimensione (e quindi il numero dei parametri), pur mantenendo informazioni sulle *feature*; rende la rete neurale tollerante alle traslazioni.
Anche per ciò che concerne questo tipo di livello, vengono definiti un *kernel* di dimensione $\boldsymbol{K} \times \boldsymbol{K}$ ed uno *stride* $\boldsymbol{S}$, utilizzati per suddividere l'input e

Input

3	2	0
4	10	8
9	-2	3

4	5	-6
7	11	2
-3	1	0

Filtro 1

1	-2
0	1

2	-1
-1	0

Bias = -5

Filtro 2

2	0
0	2

1	-1
1	-1

Bias = 4

Output

0	10
-17	11

25	44
0	40

Figura 1.9: Esempio di una convoluzione: si ha un input $3 \times 3 \times 2$ al quale vengono applicati due filtri $2 \times 2 \times 2$ (ogni tabella di una colonna indica un livello di profondità). Si ha un output di dimensione $2 \times 2 \times 2$, data dal numero di filtri (2), dall'uso dello *stride* pari ad 1 e al *padding* pari a 0.

applicare una funzione ai valori di ogni sezione. Ne esistono di vari tipi, tuttavia quelli più adoperati risultano essere l'*average pooling* ed il *max pooling* che calcolano rispettivamente un valore medio ed un valore massimo. Tali operazioni sono applicate per ogni livello in profondità, in modo indipendente, lasciando questa dimensione invariata tra l'input e l'output. A differenza dei livelli convoluzionali, quelli di *pooling* non creano nuovi pesi e *bias* poichè si limitano a calcolare una funzione prefissata sull'input.
In generale, dato un volume di dimensione $\boldsymbol{L_1} \times \boldsymbol{A_1} \times \boldsymbol{P_1}$, la larghezza $\boldsymbol{K}$ del filtro ed uno *stride* $\boldsymbol{S}$, il volume in output avrà dimensioni $\boldsymbol{L_2} \times \boldsymbol{A_2} \times \boldsymbol{P_2}$, con $\boldsymbol{P_2} = \boldsymbol{P_1}$ ed $\boldsymbol{L_2}$ definito dall'equazione 1.7 (in modo analogo anche $\boldsymbol{A_2}$).

$$L_2 = \frac{L_1 - K}{S} + 1 \tag{1.7}$$

1.4.3 Fully Connected Layer

Come per il Multilayer Perceptron, un neurone di un *fully connected layer* ha connessioni con tutti i neuroni del livello precedente, ed eventualmente con quello successivo. In altri termini, si crea una visione d'insieme di tutte le *feature* acquisite nel corso dei livelli convoluzionali e di *pooling*, per poi

Input

4	2	3	7
5	1	8	6
-4	-3	9	-8
2	1	5	2

Output (Max Pool)

5	8
2	9

Output (Avg Pool)

3	6
-1	2

Figura 1.10: Esempio di *pooling*: si ha un input $4 \times 4 \times 1$ al quale si applica un *kernel* 2×2 ed un output di dimensione $2 \times 2 \times 1$, utilizzando uno *stride* pari a 2. Sono mostrati sia il *max pooling* che l'*average pooling*.

essere mappate con una distribuzione di probabilità, al fine di risolvere un problema di classificazione.

1.5 Training

Allenare una rete neurale consiste nell'individuazione dei giusti valori numerici da assegnare ai pesi, così da ottenere una corrispondenza tra le classificazioni previste e quelle reali. Inizialmente, essi vengono scelti in modo casuale ed è necessario reperire un meccanismo tale da suggerire alla rete neurale il margine d'errore ed il modo in cui correggere la previsione. I migliaia di esempi richiesti sono utili proprio per questo scopo: per ognuno di essi, infatti, viene calcolata una sorta di funzione di costo media rispetto a tutte le possibile classi che la rete deve riconoscere oltre a tutti gli esempi sui quali viene effettuato l'allenamento. L'obiettivo è quello di trovare, appunto, i pesi che rendono minimo l'errore calcolato da tale funzione. Man mano che si ottengono queste informazioni, le opportune "modifiche" vengono applicate a ritroso nella rete neurale; per questo motivo, si parla di *backpropagation*. Calcolare la funzione di costo, esaminando tutte le migliaia di esempi, ed applicando le correzioni attraverso la *backpropagation*, è un'operazione computazionalmente molto costosa. Nella pratica quindi, si predilige l'uso di funzioni quali Stocastic Gradient Descent (SGD), basata su un processo iterativo che considera sottoinsiemi di esempi.

Un allenamento di questo tipo viene definito *supervised*, in quanto si utilizzano dati conosciuti (le reali classificazioni) per poter discriminare se una previsione è corretta o meno e modificare la rete di conseguenza.

1.6 Stato dell'Arte

Saranno presentate di seguito alcune reti neurali convoluzionali considerate da tempo lo stato dell'arte per la classificazione di immagini, insieme a qualche metrica per compararne le performance.
È doveroso osservare, tuttavia, che alcune di esse sono state recentemente superate da nuove reti, sebbene queste ultime non siano state prese in considerazione, in quanto non attualmente realizzabili attraverso le tecnologie iOS alla base di PillRecogNet.
Data una rete neurale che computa una distribuzione di probabilità per un problema di classificazione, si definisce **top-1 error** la percentuale delle volte in cui la classe prevista con la probabilità maggiore non corrisponde a quella reale. Si definisce, invece, in maniera simile, **top-5 error** la percentuale di volte in cui la classe reale non rientra nelle prime 5 previsioni. Tali concetti possono essere espressi anche sotto forma di percentuali di precisione.

1.6.1 ImageNet

ImageNet [6, 7] è uno dei più grandi dataset di immagini annotate ad alta risoluzione esistenti, impiegato prevalentemente per la ricerca nel campo della classificazione di immagini e della rilevazione di oggetti in esse. Nato come un progetto della Princeton University, si basa sulla gerarchia WordNet [8], sebbene vengano considerati quasi esclusivamente solo i nomi. Esso conta più di 14 milioni di immagini, suddivise in più di 20000 classi, con una media di circa 1000 immagini per classe. L'aspetto caratterizzante di questo dataset è quello di essere alquanto generico, rappresentando soggetti anche molto diversi tra loro (dai mammiferi, all'arredamento, agli strumenti musicali).
Su questa base, per spingere la ricerca nel campo della Visione Artificiale, nel

2010 è nata l'*ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) [9, 10], ovverosia una competizione che ha lo scopo di comparare e valutare gli algoritmi di classificazione e rilevazione delle immagini e di condividerne le scoperte ed i progressi scientifici.

1.6.2 GoogLeNet / Inception V1

Sviluppata da Szegedy et al. [11] di Google, ha vinto l'ILSVRC 2014 ottenendo un *top-5 error* del $6,67\%$ nella classificazione. Ispirata dal lavoro di Krizhevsky et al. [3], Lin et al. [12] e dedicata alla LeNet di Yann LeCun [2], questa CNN da 22 livelli ha guadagnato la vittoria prestando una maggiore attenzione all'efficienza ed alla relazione tra algoritmi, all'architettura della rete e alle risorse necessarie. È stata altresì sviluppata una nuova versione del modulo Inception, la quale ha permesso di ottenere esiti più accurati ma con una quantità notevolmente inferiore di parametri da allenare.

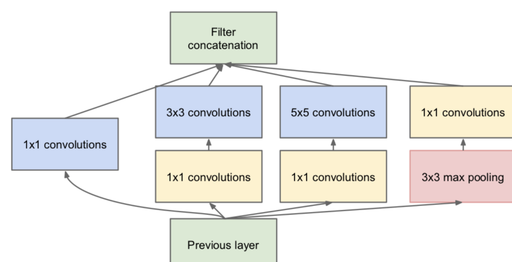


Figura 1.11: Modulo Inception utilizzato in GoogLeNet[6]

1.6.3 Inception V3

Inception V3 [13] (2015) risulta essere la composizione di alcune sperimentazioni applicate alla struttura di GoogLeNet. Precisamente, grazie alla fattorizzazione dei filtri in dimensioni minori, ad un'estesa riduzione delle dimensioni delle *feature map* e all'uso della Batch Normalization [14] per i classificatori ausiliari, è stato possibile ottenere un modello da 48 livelli tra i più efficienti esistenti, in grado di raggiungere un *top-1 error* del $21,2\%$ ed

[6]Immagine proveniente da [11]

un *top-5 error* del $5,6\%$ sui dati di validazione dell'ILSVRC 2012.
Ciò nonostante, la complessa struttura alla base di questa rete neurale la rende difficile da modificare o, tutt'al più, è fondamentale una considerevole attenzione al fine di evitare di vanificare i guadagni ottenuti in termini di costo computazionale.

1.6.4 VGGNet

VGGNet [15], sviluppata dal Visual Geometry Group dell'University of Oxford, nasce come una ricerca sull'influenza della profondità nelle reti neurali convoluzionali. L'esito consiste in una rete neurale proposta in 6 diverse configurazioni, di cui due, nello specifico, hanno registrato i migliori risultati (D ed E) e, conseguentemente, sono state rese pubbliche. Delle due, la configurazione D viene anche chiamata VGG16, in seguito alla presenza di 13 livelli convoluzionali e 3 *fully connected*. Proposte all'ILSVRC 2014, hanno registrato un *top-1 error* del $24,7\%$ ed un *top-5 error* del $7,3\%$.
Sebbene i risultati siano leggermente distanti dalla più recente Inception V3 e nonostante VGGNet necessiti di un numero più elevato di parametri, essa vanta una struttura particolarmente semplice che si sviluppa in profondità, rendendone la modifica più agevole rispetto alle reti neurali basate sul modulo Inception. Infatti, possiamo individuare in VGGNet, nella configurazione D, la base del lavoro svolto in questa tesi per creare PillRecogNet.

ConvNet	**Anno**	**Parametri**	**Top-1 Error**	**Top-5 Error**
AlexNet [3]	2012	60M	$38,1\%$	$16,4\%$
GoogLeNet [11]	2014	5M	-	$6,67\%$
VGGNet [15]	2014	133M - 144M	$24,7\%$	$7,3\%$
Inception V3 [13]	2015	25M	$21,2\%$	$5,6\%$

Tabella 1.1: Confronto delle reti neurali discusse

Capitolo 2

PillRecogNet

In questo capitolo verranno descritte le scelte implementative ed i meccanismi utilizzati per poter creare una rete neurale convoluzionale in grado di riconoscere determinati medicinali a partire dalla foto di una pillola.
Si presenteranno inoltre, alcuni risultati ottenuti e, in particolar modo, i metodi di risoluzione di alcune problematiche.
Come già spiegato nel capitolo precedente, al fine di creare una ConvNet in grado di risolvere uno specifico problema, è fondamentale un cospicuo numero di esempi in aggiunta ad un'elevata capacità computazionale. Sovente, tali requisiti possono non essere soddisfatti ed è dunque di primaria importanza valutare metodi alternativi per l'ottenimento di risultati analoghi: si discuterà infatti, di *Transfer Learning* e di come esso possa contribuire ad ottenere esiti rilevanti, nonostante l'assenza di sufficiente materiale per il *training*.

2.1 Creazione del Dataset

Una delle principali difficoltà incontrate è sicuramente la mancanza di un dataset che rappresentasse il dominio di lavoro, da utilizzare come base per l'allenamento della rete neurale. A causa della complessità e della specificità degli oggetti da riconoscere, il numero utile di esempi potrebbe essere nell'ordine delle migliaia; per giunta, ognuno di esso deve essere annotato,

azione che comporta, quindi, il coinvolgimento di soggetti esterni aventi la conoscenza adeguata al dominio in esame.
Nel caso delle pillole mediche, tuttavia, è stata fatta un'assunzione che ha in parte consentito di sopperire a tale mancanza: in linea generale, si tratta di oggetti non particolarmente complessi da riconoscere in quanto, già semplici parametri (es. forma, colore o dimensione) sono in grado di fornire una certa discriminazione. Oltre a ciò, era necessaria una quantità minima di materiale tale da permettere l'inizio del progetto.
Per queste ragioni, è stato creato un dataset di 732 foto, organizzato nei gruppi *training*, *validation* e *test*, con una proporzione di, pressappoco, 70/15/15. Ognuno di essi è a sua volta, suddiviso in 12 classi, una per ogni medicinale che la rete neurale deve essere in grado di classificare.

Figura 2.1: Alcune esempi utilizzate per il *training*

La divisione del dataset in gruppi è un passo fondamentale per l'allenamento di una CNN: infatti, le immagini di *training* sono il mezzo tramite cui essa apprende, mentre quelle in *validation* fungono da riscontro per valutare e modificare i parametri durante il *training* ed infine, le immagini di *test* sono quelle su cui viene realmente valutata l'accuratezza, in quanto si tratta di foto mai "viste" dalla ConvNet, indispensabili per poter valutare la capacità di generalizzazione su nuovi dati.

In aggiunta, sono state utilizzate tecniche di *image augmentation*, ossia degli algoritmi applicati alle fotografie per modificarle ed incrementare la dimensione del dataset. Tale scelta è stata anche dettata dal fatto che le trasformazioni in questione aiutano la rete neurale a generalizzare meglio i contenuti

da riconoscere, dacché si tiene conto di traslazioni, rotazioni, zoom ed altre manipolazioni affini, come argomentato in [16, 17, 18].
In ultimo, tutte le foto sono state preprocessate tagliando una sezione quadrata di 1120×1120 pixel raffigurante quasi esclusivamente la pillola e riducendo quindi, eventuali elementi di disturbo che potrebbero trarre in inganno la rete neurale durante l'allenamento.

Figura 2.2: Alcune immagini a cui sono state applicate le manipolazioni

2.2 Strumenti di Sviluppo

La crescita d'interesse nei confronti del Machine Learning, del Deep Learning e delle reti neurali ha portato alla nascita di svariati framework e librerie volti a semplificare l'approccio verso tali strumenti. Tra i più adoperati troviamo TensorFlow [19], sviluppato dal Google Brain Team, Theano [20], Caffe [21] e Keras [22], tutti *open source* e mantenuti insieme alla comunità; in realtà, i primi due sono specializzati nella manipolazione dei *tensori*, ossia una generalizzazione algebrica utile a rappresentare i dati elaborati dalle reti neurali.
Il progetto su cui è basata questa tesi è stato svolto impiegando Keras con TensorFlow come *backend*, in quanto esso permette un livello di astrazione più elevato, rendendo possibile creare reti neurali con poche righe di codice Python. Nello specifico, si è partiti dalla VGG16, i cui livelli finali sono stati sostituiti da una piccola rete neurale per effettuare il *transfer learning* (le strutture sono mostrate in dettaglio nell'Appendice A).

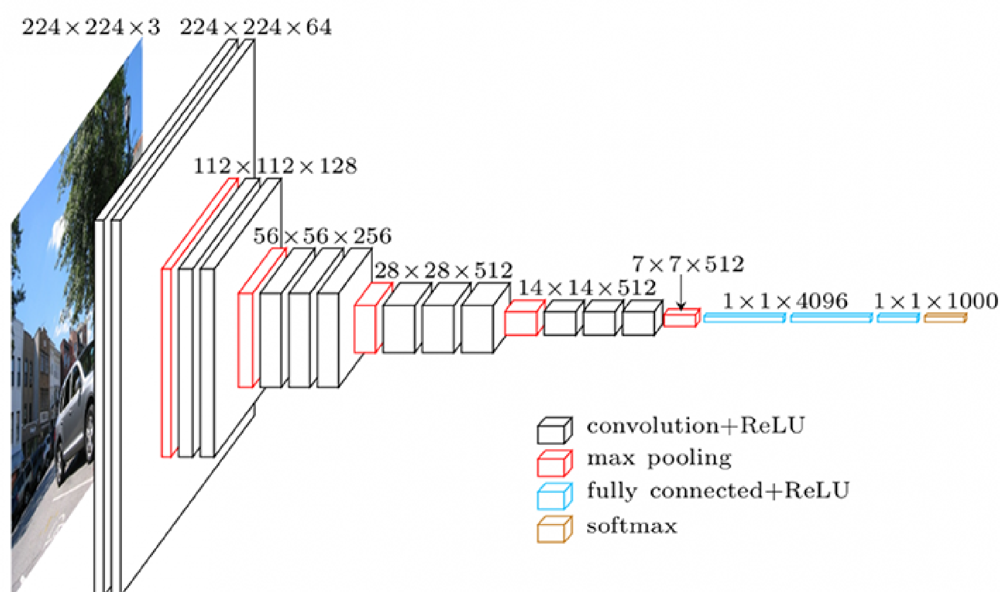


Figura 2.3: Struttura predefinita della VGG16

Inizialmente, le varie operazioni di *training* sono state effettuate su un MacBook Pro 2017 con il solo ausilio della CPU, necessitando di circa 12/13 ore per la computazione. Data la necessità di dover eseguire più allenamenti al fine di individuare i parametri migliori, si è optato di trasferire il *training* su un'istanza `p2.xlarge` fornita da Amazon AWS[1], dotata di una GPU NVIDIA Tesla K80 e ideata specificamente per il supercalcolo. Grazie a tale variazione, la quantità di tempo investita nell'effettuare questo procedimento si è drasticamente ridotta a poche ore, o meno.
Infine, ottenuta la rete neurale, si è passati all'implementazione della stessa in un'applicazione iOS, attraverso Xcode: visto che le tecnologie utilizzate si interfacciano direttamente con la GPU dello smartphone, è stato fondamentale sfruttare la licenza da Apple Developer[2] per poter testare il progetto su un dispositivo reale.

[1] `https://aws.amazon.com/it/`
[2] `https://developer.apple.com/programs/`

2.3 Transfer Learning

Fino a pochi anni fa non si era a conoscenza del motivo per il quale una rete neurale convoluzionale fosse così precisa ed efficiente nel riconoscere le immagini, tanto da essere trattate come scatole nere funzionanti quasi per "magia". Realisticamente, ciò non è possibile ma, in seguito al lavoro svolto da Zeiler et al. [23], si è raggiunto un maggiore grado di consapevolezza sul loro funzionamento interno. Attraverso alcune tecniche di visualizzazione, è stata infatti dimostrata una relazione tra i livelli e ciò che sono in grado di apprendere: man mano che si scende in profondità, si ha una maggiore capacità di discriminazione, relativamente al contenuto di un'immagine. Ad esempio, si parte dal riconoscere elementi semplici, come contorni ed angoli, fino a complesse composizioni di queste nozioni.

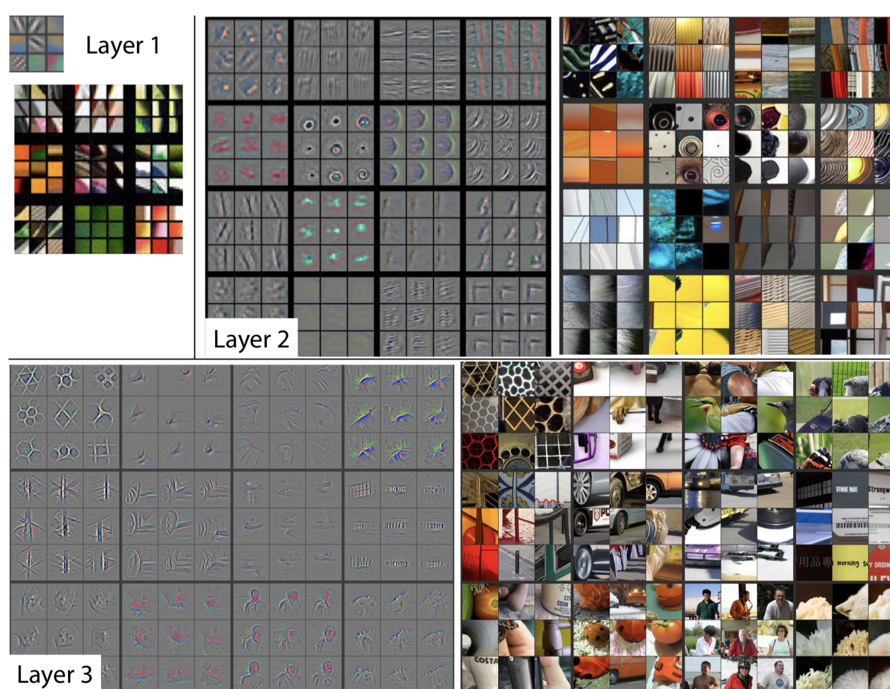
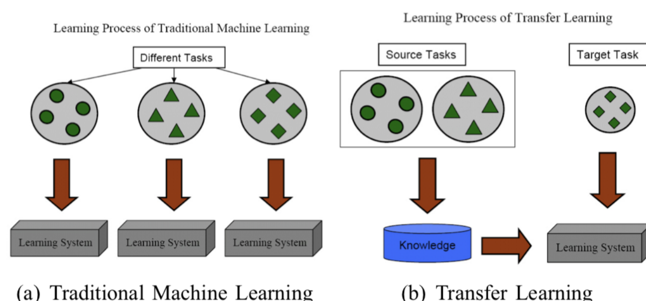


Figura 2.4: Esempio dell'evoluzione delle *feature* riconosciute da una CNN[3]

[3]Immagine proveniente da [23]

Una caratteristica interessante, e sotto alcuni aspetti sorprendente, è che questo comportamento sembra avvenire nella quasi totalità degli allenamenti basati su immagini, indipendentemente dalla struttura della rete, dagli algoritmi oppure dal dataset utilizzato. Le *feature*, difatti, sono solo il risultato delle elaborazioni svolte dai neuroni; in altre parole, esse dipendono principalmente dai pesi e dai *bias*.

Contestualmente a tali esiti, ci si è anche interrogati sulla generalità dei parametri di una rete neurale, così come sulla loro possibilità di essere riutilizzati: per *transfer learning* si intende proprio la possibilità di adattare e trasferire i pesi, al fine di poter usare nuovamente la "conoscenza" per perseguire molteplici obiettivi diversi.



(a) Traditional Machine Learning (b) Transfer Learning

Figura 2.5: Rappresentazione dell'idea alla base del *transfer learning*[4]

Il procedimento appena descritto avviene, solitamente, in due fasi: dapprima, quasi tutti i livelli della rete neurale vengono allenati su un dataset molto grande e generico, tale da consentire l'acquisizione di nozioni globali; in seguito, si utilizzerà il dataset specifico per il training dei *layer* rimanenti, decidendo o meno di far propagare gli errori attraverso il *fine tuning*. Di questo e del problema dell'*overfitting* si accennerà nelle sezioni successive.
Pan at al. [24] hanno approfondito il tema del *transfer learning*, applicandolo ad esempi di classificazione, regressione o clustering, oltre a mostrare quanto possa essere utile per sopperire alla mancanza di dati nel dominio in esame.

[4]Immagine proveniente da [24]

Se da un lato è possibile riutilizzare la conoscenza di una rete neurale, è comunque necessario comprenderne la specificità, soprattutto nel caso in cui il problema finale è poco affine a quello di partenza.

Grazie a Yosinski et al. [25] è stato sperimentalmente dimostrato[5] che i livelli iniziali di una ConvNet si occupano di riconoscere *feature* molto generiche mentre quelle risultanti dai livelli finali sono più specializzate e correlate al dataset. Inoltre, sotto alcune condizioni, il trasferimento dei pesi può comportare un grande miglioramento della capacità di generalizzazione, aumentando l'accuratezza generale della CNN.
Il lavoro svolto si è quindi basato su queste premesse: partire dalla VGG16 già allenata su ImageNet, estrarne i parametri e riutilizzarli in PillRecogNet, la rete neurale strutturalmente simile ma pensata per la classificazione delle pillole mediche. La struttura finale in dettaglio è presente nell'Appendice A.

Uno degli aspetti vantaggiosi nell'utilizzare Keras è che molte delle reti allo stato dell'arte, con i relativi pesi calcolati su ImageNet, sono già implementate all'interno del framework, consentendone l'uso con una semplice riga di codice Python e permettendo all'utilizzatore di concentrarsi sulle personalizzazioni da mettere in pratica.

```
1 from keras.applications import VGG16
2
3 img_width, img_height = 224, 224
4 model = VGG16(weights="imagenet",
     include_top=False, input_shape=(img_width,
     img_height, 3))
```

Listato 2.1: Creazione della VGG16 con i parametri ma senza i livelli *fully connected*

[5] https://github.com/yosinski/convnet_transfer

2.3.1 Preprocessing

Come accennato in precedenza, per far sì che il *transfer learning* vada a buon fine, bisogna rispettare alcune condizioni: il dataset iniziale e quello finale non devono essere troppo diversi tra loro e devono condividere le stesse operazioni di *preprocessing*. In altre parole, le immagini del nuovo dataset devono essere elaborate nello stesso modo in cui lo sono state quelle iniziali, in modo da garantire che i parametri trasferiti siano consistenti.
In particolare, come evidenziato nelle specifiche di VGG16, l'unica elaborazione da effettuare consiste nel sottrarre i valori medi dei canali R, G e B calcolati su ImageNet prima del *training*.

```
1 import numpy as np
2
3 # x è un tensore che rappresenta un'immagine
4 def meanSubtraction(x):
5     x = x.astype(np.float32)
6     means = np.array([123.68, 116.779, 103.939],
   dtype=np.float32).reshape((1,1,3))
7     x -= means
8     return x
```

Listato 2.2: Funzione di *preprocessing* applicata ad ogni immagine del nuovo dataset per sottrarre i valori RGB medi

In seguito a tale manipolazione, i valori che un pixel può assumere sono più o meno centrati rispetto allo zero. Essi vengono scalati con un fattore 255, in modo da lavorare con numeri reali compresi tra $[-1.0, 1.0]$.
Così facendo, è possibile usufruire di un *learning rate* molto piccolo, tale da permettere alla rete neurale di apprendere ma, allo stesso tempo, di non intaccare pesantemente i parametri ottenuti dall'allenamento su ImageNet.

2.3.2 Feature Extraction

Prima di procedere al *training* vero e proprio sul dataset dei medicinali, è necessario un ulteriore passo: la ConvNet (Appendice A.2) che sostituisce i livelli *fully connected* della VGG16 deve essere inizializzata.
Si potrebbe pensare che assegnare valori casuali ai relativi pesi e *bias* possa essere una soluzione ma, in verità, non si tratta di un metodo efficace ed attendibile, soprattutto nel caso di reti neurali dotate di una elevata quantità di livelli (come mostrato in [25]). Ciò è principalmente dovuto al fatto che la funzione di costo alla base della *backpropagation* potrebbe calcolare errori troppo elevati, causando una netta modifica dei parametri iniziali e, quindi, intaccare la conoscenza trasferita.
Si è quindi pensato di utilizzare i soli livelli convoluzionali come *feature extractor*, ovverosia le nuove immagini sono valutate dalla rete principale basata su VGG16 così da ottenere i parametri che costituiranno le *feature map* dell'ultimo livello convoluzionale. In questo modo, l'inizializzazione avviene con pesi più coerenti, permettendo alla ConvNet aggiuntiva di raggiungere buoni risultati di classificazione.

2.3.3 Fine Tuning

Al fine di migliorare ulteriormente l'accuratezza di PillRecogNet, è stata eseguita la tecnica del *fine tuning*, che consiste nel permettere la propagazione degli errori, e quindi l'aggiustamento dei pesi, anche ai livelli che precedono quelli personalizzati. Tuttavia, è necessario procedere con cautela per non incorrere nel problema dell'*overfitting*, ovvero quando una rete neurale acquisisce caratteristiche troppo correlate al dataset su cui è stata allenata, impedendole di riuscire nella classificazione di nuovi dati.
Considerando quindi il limitato numero di immagini a disposizione, si è deciso di applicare tale tecnica solo agli ultimi tre livelli convoluzionali (quelli appartenenti a `block5` nel dettaglio presente nell'Appendice A.1).
Oltre all'*image augmentation*, di cui si è trattato in precedenza, un ulteriore

meccanismo è stato utilizzato al fine di limitare tale fenomeno: il *dropout* [26], cioè una tecnica di regolarizzazione che consiste nell'associare una probabilità ai neuroni, attraverso cui decidere se alcuni di essi debbano essere disattivati o meno durante il *training*, in modo casuale. In altre parole, è come se si stesse campionando la rete neurale in un numero esponenziale di reti più semplici, la cui previsione finale sarà data dalla media dei risultati. Va notato, però, che tale comportamento è applicato solo durante la fase di allenamento e non durante l'inferenza, ossia la semplice esecuzione della CNN allenata per ottenere la classificazione di un'immagine.

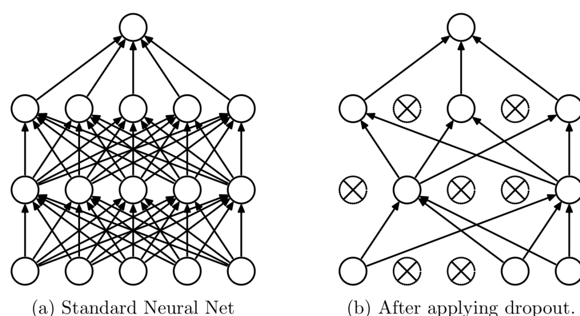


Figura 2.6: Esempio di rete neurale a cui è stato applicato il *dropout*[6]

2.4 Statistiche e Risultati

Il dettaglio di PillRecogNet è mostrato nell'Appendice A.1; la parte fondamentale, specializzata nel riconoscimento dei medicinali, è data dalla piccola ConvNet aggiunta ai livelli convoluzionali ed è descritta nell'Appendice A.2. Di seguito verranno mostrate le scelte effettuate, riguardanti l'allenamento, i relativi risultati ed un esempio di classificazione.

```
1 from keras.models import Sequential
2 from keras.layers import Dropout, Flatten, Dense
3
```

[6]Immagine proveniente da [26]

```
4 top_model = Sequential()
5 top_model.add(Flatten(input_shape =
    base_model.output_shape[1:]))
6 top_model.add(Dense(512, activation="relu"))
7 top_model.add(Dropout(0.5))
8 top_model.add(Dense(num_classes,
    activation="softmax"))
```

Listato 2.3: Definizione della rete neurale aggiunta ai livelli convoluzionali di VGG16 utilizzando Keras. L'intero codice in dettaglio è mostrato nell'Appendice B.

Le strategie di *training* utilizzate sono leggermente diverse, sebbene entrambe si basino sullo Stocastic Gradient Descent per la *backpropagation*.
Nel caso dell'estrazione delle *feature* (spesso indicate anche come *bottleneck*, data la forma a "collo di bottiglia" della struttura nel punto in cui terminano i livelli convoluzionali), non è stata applicata l'*image augmentation* allo scopo di ottenere valori il più possibile coerenti con le immagini del dataset; il *learning rate* utilizzato è stato fissato a 10^{-3}, con un numero di epoche (ovvero il numero di volte in cui l'intero dataset di *training* viene analizzato) pari a 100. L'andamento dell'accuratezza e dell'errore in questa fase sono mostrati nelle Figure 2.8(a) e 2.8(b), rispettivamente.
Nel caso del *fine tuning* invece, le differenze sostanziali consistono nell'aver applicato l'*image aumentation* al dataset di *training*, un numero di epoche inferiore (50) oltre ad un *learning rate* dinamico. Quest'ultimo difatti, è stato dimezzato ogni 17 epoche circa (corrispondenti ad 1/3 del numero totale), partendo da 10^{-3} e fino a $2,5 \times 10^{-4}$ (Figura 2.7).
La scelta di utilizzare un *learning rate* più basso durante il *fine tuning* è dovuta all'idea che i pesi nella rete neurale risultano essere già abbastanza buoni e si vuole evitare che essi vengano distorti. Anche per questa fase, accuratezza ed errore sono mostrati nelle Figure 2.8(c) e 2.8(d).

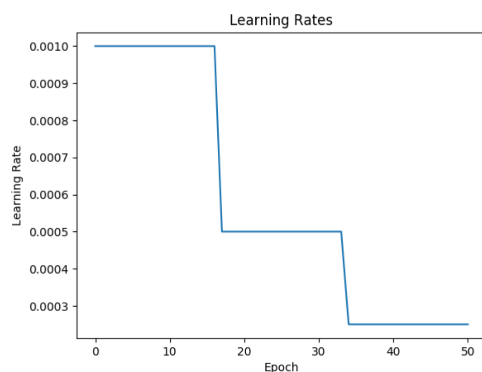
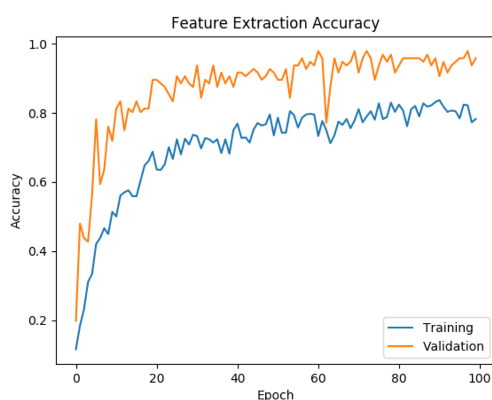
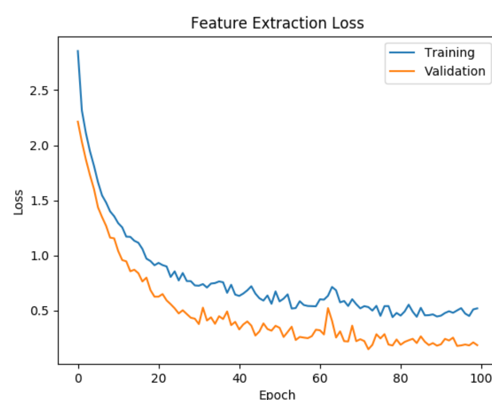


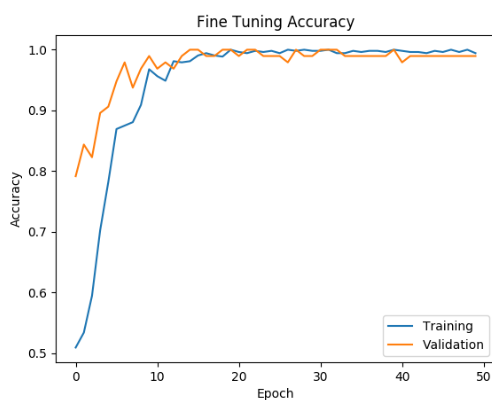
Figura 2.7: Andamento del *learning rate* durante il *fine tuning*



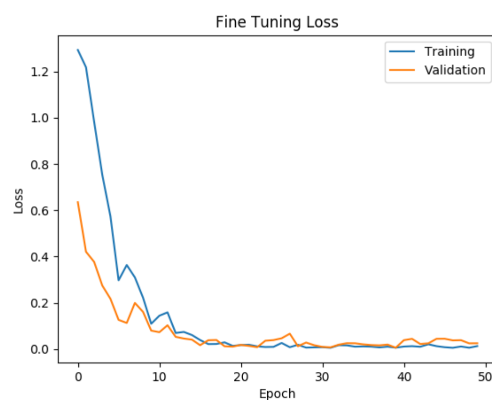
(a) Accuratezza dopo l'estrazione delle *feature*



(b) Errore dopo l'estrazione delle *feature*



(c) Accuratezza dopo il *fine tuning*



(d) Errore dopo il *fine tuning*

Figura 2.8: Statistiche relative al *training*

Per poter ottenere una distribuzione di probabilità, l'output della rete neurale viene elaborato dalla funzione *softmax* che si occupa di trasformare il risultato in uno nuovo che abbia la somma degli elementi pari ad 1. Così facendo, vengono messe in evidenza le classi che hanno probabilità maggiore.

Come si evince dai grafici e dalla Tabella 2.1, PillRecogNet sembra comportarsi tendenzialmente bene nella classificazione, soprattutto considerando le ridotte dimensioni del dataset di *training*. Inoltre, la differenza tra l'accuratezza di *training* e di validazione è principalmente dovuta all'utilizzo del *dropout* durante l'allenamento.

Dataset	**Totale Immagini**	**Corrette**	**Top-1 Accuracy**
Validation	96	95	$98,96\%$
Test	108	106	$98,15\%$

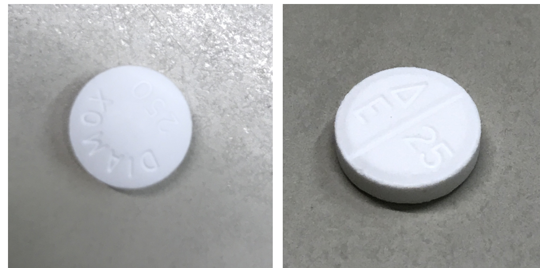
Tabella 2.1: Risultati dell'accuratezza sui sottoinsiemi del dataset

Esempi di classificazione corretta

(a) Reale: $90,18\%$ (b) Reale: $99,97\%$ (c) Reale: $98,86\%$

Figura 2.9: Sono mostrate tre immagini che la rete neurale riconosce correttamente; il valore in percentuale indica l'accuratezza della prima classe riconosciuta

Esempi di classificazione errata



(a) Previsto: $87,44\%$
Reale: $12,56\%$

(b) Previsto: $99,98\%$
Reale: $1,39 \times 10^{-7}\%$

Figura 2.10: Sono mostrate due immagini che la rete neurale non riesce a riconosce correttamente; il primo valore in percentuale indica l'accuratezza della classe prevista mentre il secondo, l'accuratezza della classe reale

Capitolo 3

Applicazione iOS

Nel seguente capitolo sarà effettuata una breve panoramica sulle tecnologie iOS che permettono di eseguire reti neurali in uno smartphone, analizzandone le differenze, i requisiti e le scelte implementative che hanno portato alla creazione di PillRecogNet App.

3.1 ConvNet in Ambito iOS

Nonostante la presenza di strumenti che permettono la creazione e l'allenamento di reti neurali su un computer, non è scontato averne di analoghi che siano concepiti per essere utilizzati su uno smartphone. Durante la Apple WorldWide Developer Conference[1] 2016 e con la presentazione di iOS 10, si è assistito ad un cambiamento di rotta, grazie al quale sono stati forniti i primi framework che permettono l'integrazione di una rete neurale in ambito mobile, almeno per ciò che concerne i dispositivi dell'ecosistema Apple. Considerando l'elevato numero di operazioni svolte da una CNN, l'obiettivo primario degli strumenti in questione è quello di massimizzare ed ottimizzare le prestazioni e lo sfruttamento dell'hardware. Per tale ragione infatti, il codice risulta essere spesso verboso e di più basso livello rispetto ad altri framework iOS, ad esempio UIKit. Un ulteriore aspetto da valutare riguarda

[1] `https://developer.apple.com/wwdc/`

l'impossibilità di effettuare il *training* sul dispositivo mobile, bensì solo l'inferenza. Le API (*Application Programming Interface*) di cui si approfondirà in seguito sono, nell'insieme, soluzioni native ideate principalmente per l'implementazione di una ConvNet, in quanto si è scelto di non considerare *porting* alternativi o anche metodi che non garantissero una minima scalabilità del codice per l'applicazione finale.

3.1.1 Basic Neural Network Subroutines (BNNS)

BNNS [27] consiste in un insieme di funzioni appartenenti al framework Accelerate, il cui scopo principale è quello di implementare le operazioni matematiche alla base di una rete neurale, come i prodotti matriciali. Il codice risultante, difatti, più che descrivere strutturalmente una CNN, ne indica le manipolazioni da applicare ai dati in input. Essendo un'API ottimizzata per l'esecuzione sulla CPU, risulta essere compatibile con la maggior parte dei dispositivi iOS, macOS, watchOS e tvOS. In particolare, è necessario iOS 10.0, o superiore, per l'utilizzo in un'applicazione su smartphone o tablet e permette di elaborare i dati attraverso livelli convoluzionali, di *pooling* (*max* o *average*) o *fully connected*. Infine, nonostante siano funzioni native in Swift, la loro scrittura è molto *C-like* e quindi verbosa.

```
1  BNNSConvolutionLayerParameters layer_params;
2  bzero(&layer_params, sizeof(layer_params));
3  layer_params.k_width = 5;   // convolution kernel width: 5 pix
4  layer_params.k_height = 5;   // convolution kernel height: 5 pix
5  layer_params.in_channels = i_desc.channels;   // input channels
6  layer_params.out_channels = o_desc.channels;  // output channels
7  layer_params.x_stride = 1;    // x stride: 1 pix
8  layer_params.y_stride = 1;    // y stride: 1 pix
9  layer_params.x_padding = 0;   // x padding: 0 pix
10 layer_params.y_padding = 0;   // y padding: 0 pix
11 //      . . .
12 BNNSFilterParameters filter_params;
13 bzero(&filter_params, sizeof(filter_params));
14
15 BNNSFilter filter = BNNSFilterCreateConvolutionLayer(&i_desc, &o_desc,
       &layer_params, &filter_params);
```

Listato 3.1: Descrizione di un livello convoluzionale tramite BNNS

3.1.2 Metal Performance Shaders (MPS)

I Metal Performance Shaders [28] fanno parte di Metal [29], ovvero il framework specializzato nell'elaborazione grafica 2D e 3D in ambito iOS, macOS e tvOS. Nello specifico, essi permettono lo sfruttamento della GPU per l'esecuzione di algoritmi computazionalmente dispendiosi, grazie al parallelismo. Sebbene siano disponibili agli sviluppatori già da iOS 9, è con iOS 10 che è stato possibile realizzare reti neurali convoluzionali strutturalmente complesse, in aggiunta ad altre operazioni quali l'*image processing* e il prodotto tra matrici in modo efficiente. In particolare, sono disponibili livelli convoluzionali, di *pooling* (*max* e *average*), *fully connected* e di normalizzazione, oltre alla facoltà di poter creare livelli e filtri personalizzati, questi ultimi tramite il Metal Shading Language [30] di cui si accennerà in seguito.
A differenza delle BNNS, gli MPS risultano essere molto più performanti, soprattutto nel caso in cui la ConvNet abbia una quantità elevata di livelli. Tuttavia, tutte le ottimizzazioni messe in atto comportano un costo per quanto riguarda la compatibilità: il loro utilizzo è permesso esclusivamente ai dispositivi iOS che abbiano un SoC (*system-on-a-chip*) Apple A8 o superiore[2], includendo gli smartphone ed i tablet dal 2014 ad oggi.
Non considerando il Metal Shading Language (molto simile al C++), l'intero codice viene scritto in Swift ma con un'astrazione leggermente superiore rispetto alle BNNS, malgrado la ripetitività di dover specificare l'ordine delle operazioni da far eseguire alla GPU.

```
1 let convDescriptor = MPSCNNConvolutionDescriptor(kernelWidth: 5,
    kernelHeight: 5, inputFeatureChannels: inputDepth,
    outputFeatureChannels: outputDepth, neuronFilter: nil)
2 convDescriptor.strideInPixelsX = 1
3 convDescriptor.strideInPixelsY = 1
4
5 let convLayer = MPSCNNConvolution(device: currentDevice,
    convolutionDescriptor: convDescriptor, kernelWeights: weights,
    biasTerms: bias, flags: .none)
```

Listato 3.2: Descrizione di un livello convoluzionale tramite MPS

[2]Tecnicamente, `GPU Family 2v3`, `GPU Family 3v2`, `GPU Family 4v1` e superiori, come mostrato in `https://developer.apple.com/metal/Metal-Feature-Set-Tables.pdf`

3.1.3 CoreML e Vision

Durante la Apple WorldWide Developer Conference 2017, sono stati presentati due nuovi framework che permettono di integrare il Machine Learning in un'applicazione: CoreML [31] e Vision [32], quest'ultimo specializzato nella Visione Artificiale. La loro introduzione ha principalmente avuto come scopo quello di mitigare la difficoltà precedentemente richiesta, dovuta alla scrittura di codice di basso livello in rapporto alla maggioranza dei framework iOS. Lo scopo finale è, effettivamente, quello di nascondere la maggior parte della complessità, così che chiunque possa implementare un modello di Machine Learning nella propria applicazione.
Il tutto si basa sull'importazione di un file contenente la descrizione del modello oltre ai relativi parametri, lasciando a CoreML il compito di caricarlo ed eseguirlo nel modo più appropriato in relazione al dispositivo in uso. In questo modo, se da un lato basta indicare il modello da caricare ed invocare un metodo per ottenere il risultato, non è possibile intervenire sul funzionamento interno, anche nel caso si usi una rete neurale creata ed allenata ad hoc.
Inoltre, l'applicazione da implementare deve richiedere almeno iOS 11 per poter usufruire di tali framework; nonostante tale versione del sistema operativo iOS sia stata rilasciata da pochi mesi ed installata da più della metà dei dispositivi compatibili in circolazione[3], risulta comunque essere un limite troppo restrittivo per il lavoro svolto, non permettendone l'impiego.

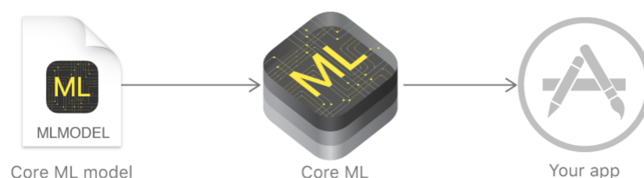


Figura 3.1: Schema di integrazione di un modello di Machine Learning con CoreML[4]

[3]Dati ufficiali da https://developer.apple.com/support/app-store/

3.2 Scelte Implementative

Come accennato in precedenza, la scelta principale che ha in parte dettato quelle successive ha riguardato la versione minima di iOS da supportare, scegliendo iOS 10.0 ed escludendo quindi, la possibilità di utilizzare CoreML e Vision. Inoltre, l'applicazione è stata pensata per essere utilizzata prevalentemente su uno smartphone o, tutt'al più, un tablet. Per questi motivi, si è deciso di implementare PillRecogNet tramite i Metal Performance Shaders, i quali mettono a disposizione strumenti più specifici per la definizione di una rete neurale complessa e l'accelerazione della GPU durante l'inferenza.

In linea generale, l'uso dei Metal Performance Shaders per le reti neurali convoluzionali avviene in modo più o meno sequenziale: si parte dal definire la CNN indicando i vari livelli, le funzioni di attivazione ed i descrittori delle immagini che la attraverseranno, per poi inizializzarla ed effettuare l'*encoding* dei dati per ottenere una classificazione. In questo contesto, per "immagine" si intende un oggetto di tipo `MPSImage`, specializzato nella rappresentazione dei dati da elaborare attraverso una ConvNet perché predisposti alla gestione di un elevato numero di canali. Inoltre, al fine di ottimizzare maggiormente l'impiego della memoria sul dispositivo, gli *encoding* intermedi delle immagini avvengono con oggetti temporanei di tipo `MPSTemporaryImage`.

```
1 // Definizione ed inizializzazione
2 let block1_conv1: PillConvolution
3 let conv_block1ImgDesc = MPSImageDescriptor(channelFormat: .float16, width:
      224, height: 224, featureChannels: 64)
4 block1_conv1 = PillConvolution(device: device, inputDepth: 3, outputDepth:
      64, parametersName: "conv1", filter: relu)
5
6 // Encoding
7 let b1c1Img = MPSTemporaryImage(commandBuffer: comBuf, imageDescriptor:
      conv_block1ImgDesc)
8 block1_conv1.encode(commandBuffer: comBuf, sourceImage: preprocessedImage,
      destinationImage: b1c1Img)
```

[4]Immagine proveniente da [31]

```
9 preprocessedImage.readCount = 0
```

Listato 3.3: Esempio di definizione, inizializzazione ed *encoding* di un'immagine in un livello convoluzionale

La funzione di classificazione si occupa, dunque, di ripetere tali operazioni per ogni livello, seguendo la struttura della rete neurale.

Preprocessing delle immagini

Anche nell'applicazione iOS è necessario che le immagini da classificare vengano preprocessate con le stesse operazioni applicate durante il *training* della rete neurale. Tuttavia, gli MPS non mettono a disposizione nessun *kernel* o livello che esegua tali operazioni ed è stato quindi necessario implementarlo. Nell'ambito del framework Metal, una filtro descrive le operazioni che la GPU deve eseguire per manipolare i pixel di un'immagine sotto forma di *texture*. In particolare, essi possono essere realizzati attraverso Metal Shading Language, un linguaggio basato su C++.
Prima che venga applicato il preprocessing, l'immagine viene caricata come una texture `RGBA8Unorm`, ad indicare che i valori dei pixel tra 0 e 255 sono normalizzati sull'intervallo $[0, 1]$ e gestiti come `float` nei livelli successivi. È per questo che, come si evince dal Listato B.9, i valori dei canali R, G e B sono riportati all'intervallo $[0, 255]$, ne viene sottratto il valore medio ed infine, vengono scalati per rispettare l'intervallo $[-1, 1]$ utilizzato durante il *training*.

Esportazione e reshape dei parametri

Una volta definita la rete neurale con gli MPS, ogni livello deve essere inizializzato con i pesi ed i *bias* calcolati durante il *training*. È stato quindi implementato uno script Python (mostrato nel Listato B.4) che carica il modello allenato e, per ogni livello, ne estrapola i parametri sotto forma di file binari, secondo la nomenclatura `conv{indice}_weights`, `conv{indice}_bias` per i livelli convoluzionali e `fc{indice}_weights`, `fc{indice}_bias` per quelli

fully connected.
Un aspetto importante da considerare è l'ordine dei valori durante l'esportazione; difatti, i livelli definiti attraverso i Metal Performance Shaders si aspettano i pesi in una determinata disposizione per poter essere rappresentati come tensori. Tale ordine è dato da:

`[outputDepth][kernelHeight][kernelWidth][inputDepth]`

mentre Keras, con TensorFlow come *backend*, prevede che essi abbiano una disposizione del tipo:

`[inputDepth][outputDepth][kernelHeight][kernelWidth]`

Per questo, i pesi sono stati appositamente manipolati prima di essere esportati, attraverso le funzioni *reshape* e *transpose* della libreria NumPy [33].

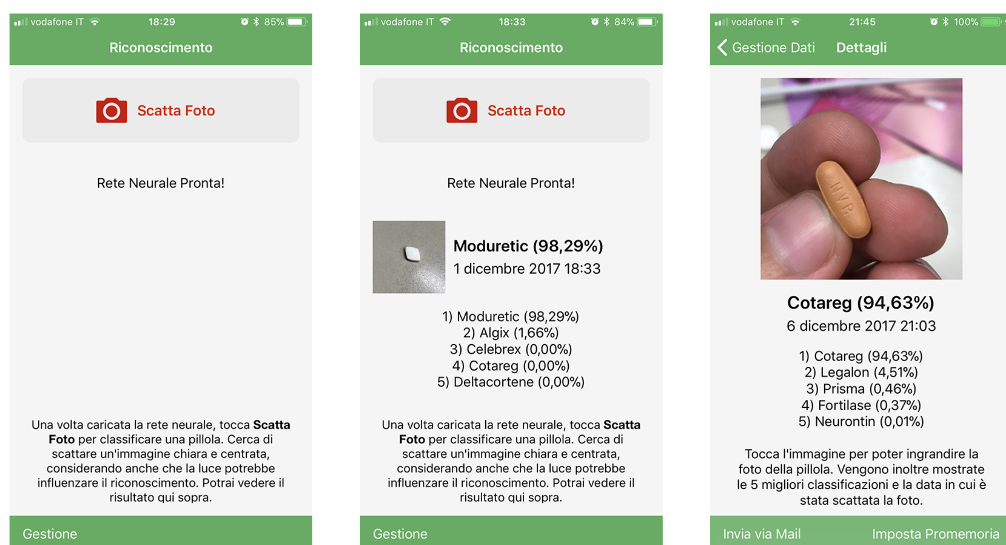
Gestione delle classi

Sebbene la rete neurale è stata allenata per classificare 12 medicinali, la gestione delle classi nell'applicazione è stata pensata in modo da permetterne una facile espansione. Infatti, tutto si basa su un file di testo con il seguente formato:

```
0|Classe 1
1|Classe 2
...
n|Classe n+1
```

Così facendo, dopo aver eseguito nuovamente il *training* su un numero maggiore di classi da riconoscere, basta aggiungere le relative righe al file di testo e modificarne il numero totale indicato nell'implementazione (mostrata nel Listato B.8). Va notato che la corrispondenza tra indice e classe deve rispettare quella ottenuta dall'allenamento (durante il *fine tuning* oppure la valutazione), per non incorrere in risultati inconsistenti.

3.3 PillRecogNet App



(a) Schermata iniziale (b) Risultato classificazione (c) Dettaglio classificazione salvata

Figura 3.2: Alcune schermate dell'applicazione iOS in uso

Oltre alla rete neurale convoluzionale, il risultato vero e proprio del lavoro svolto è l'applicazione iOS che la incorpora e ne permette l'uso in mobilità. L'obiettivo principale è quello di consentire l'inferenza, ovvero il riconoscimento dei medicinali partendo da una semplice fotografia della pillola.
Al fine di semplificarne il più possibile l'utilizzo, i parametri della rete neurale vengono caricati in modo asincrono all'apertura, disabilitando momentaneamente la possibilità di scattare una foto. In modo analogo, l'interfaccia utente viene ulteriormente disabilitata durante la classificazione, in attesa che la rete neurale completi l'elaborazione. In altre parole, la UI (*user interface*) viene automaticamente gestita in modo da permettere l'interazione con l'utente solo quando necessario (ad esempio, il pulsante per scattare una foto viene colorato di rosso o di grigio in base al suo stato).
Tra l'altro, nel momento in cui si sta per fotografare il medicinale, vengono

applicati alcuni accorgimenti per far in modo che la pillola risulti essere l'elemento in primo piano dell'immagine (utilizzando lo zoom e il ritaglio della parte centrale), così da evitare che l'elevata risoluzione possa includere elementi di disturbo per la classificazione.

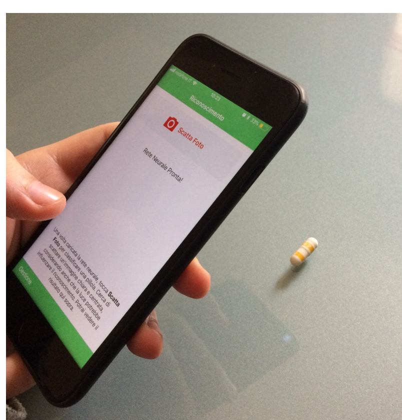
In seguito ad alcuni test effettuati su un dispositivo fisico, si evince che l'operazione più dispendiosa è sicuramente l'inizializzazione della rete neurale con il relativo caricamento dei parametri. Difatti, tale operazione necessita di circa 1,5 secondi mentre l'inferenza viene effettuata in circa 0,4 secondi.

Esperienza d'uso

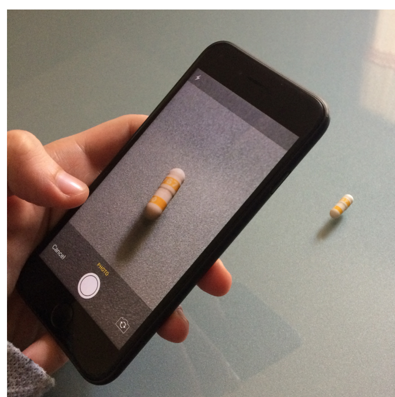
L'idea di funzionamento consiste nello scattare una foto ad un medicinale, che sia tenuto in mano, tra le dita o su una superficie, ottenerne la classificazione e memorizzarla, insieme ad altre informazioni quali data, ora, la foto in originale e i dati sull'accuratezza delle prime cinque classi ottenute. Si tiene traccia di tutti i riconoscimenti effettuati, con le relative fotografie, per permetterne una consultazione futura oppure una eventuale condivisione remota.
Per ogni classificazione effettuata, si ha la possibilità di impostare un promemoria (singolo o giornaliero), in modo tale che l'utente riceva una notifica e, quindi, assuma il medicinale. Inoltre, le informazioni relative ad ogni riconoscimento possono essere facilmente inviate tramite email.
Sebbene queste ultime funzioni abbiano un'implementazione di base, sono pensate per essere integrate ed espanse con un servizio remoto, ad esempio tramite comunicazione diretta con i medici.
Nelle figure successive si mostra l'esperienza d'uso dell'applicazione attraverso alcune fotografie.

In conclusione, possiamo affermare che quest'applicazione può essere considerata un prototipo funzionale, utile a dimostrare la possibilità di implementare una ConvNet complessa da circa 28 milioni di parametri in un piccolo

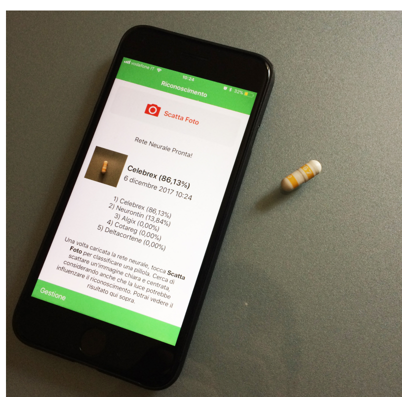
dispositivo come uno smartphone, raggiungendo comunque risultati notevoli, grazie allo sfruttamento della GPU e delle ottimizzazioni messe in atto. Per questo motivo, data la limitata disponibilità di risorse, esempi e documentazione, si è deciso di rendere il codice sorgente *open source*[5], in modo che possa essere collaborativamente migliorato o, semplicemente, di supporto a chiunque voglia perseguire un obiettivo simile su iOS 10 attraverso i Metal Performance Shaders.



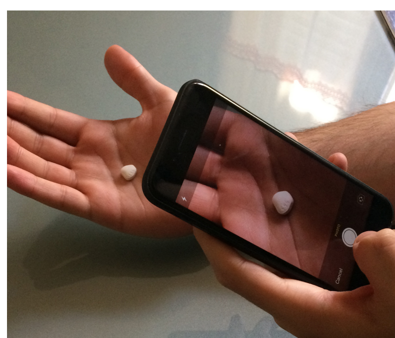
(a) Apertura dell'applicazione



(b) Scatto della foto al medicinale



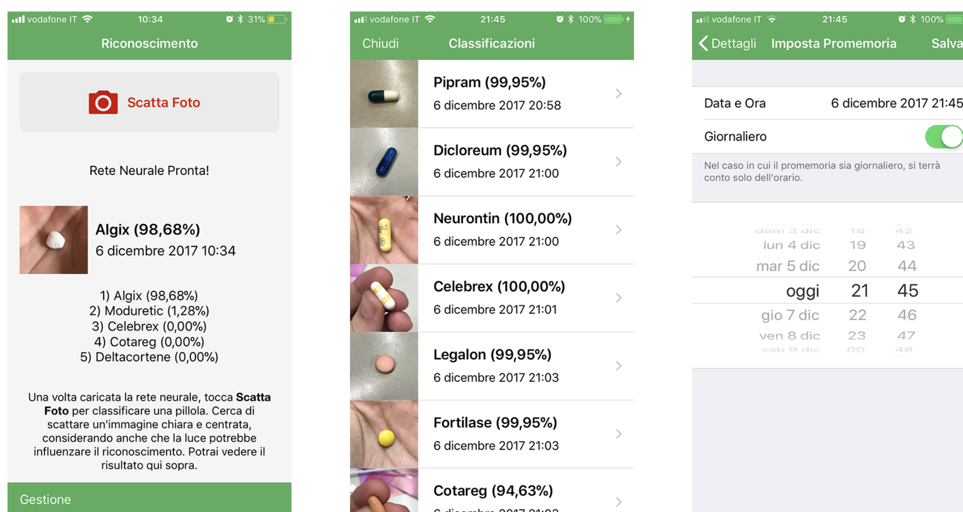
(c) Risultato della classificazione



(d) Esempio di scatto dalla mano

Figura 3.3: Alcune fotografie che dimostrano l'utilizzo reale dell'applicazione

[5]Rete neurale: `https://github.com/matteodelv/PillRecogNet`, Applicazione iOS: `https://github.com/matteodelv/PillRecogNet-iOS`



(a) Risultato riconoscimento dalla mano mostrato in Figura 3.3(d)

(b) Esempio di storico delle classificazioni

(c) Schermata di impostazione del promemoria

Figura 3.4: Sono mostrate ulteriori schermate dell'applicazione iOS mettendone in risalto alcune funzioni aggiuntive

Conclusioni

Nelle pagine precedenti è stata trattata la questione del riconoscimento di un medicinale, partendo da una sua fotografia, fino alla relativa realizzazione di un'applicazione iOS da utilizzare in mobilità.
Sebbene possa sembrare un compito non particolarmente complesso, è stato necessario partire dalle conoscenze teoriche alla base del Machine Learning e delle reti neurali convoluzionali, perché attualmente risultano essere il modello di calcolo che permette risultati migliori nella Visione Artificiale.
Si è quindi discusso di alcuni framework esistenti per la definizione ed il training di una ConvNet, di come sfruttare il *transfer learning* allo scopo di riutilizzare la "conoscenza" di una rete neurale allo stato dell'arte e di come è stato possibile specializzarla sul dominio d'interesse.
In seguito è stato trattato l'ambito *mobile*, con particolare riferimento ad iOS, mostrando brevemente alcuni dei possibili meccanismi che permettono l'implementazione di una rete neurale convoluzionale ed illustrandone i limiti.
L'applicazione realizzata, nonostante si basi su tecnologie con requisiti più o meno restrittivi, permette di sfruttare l'accelerazione della GPU per l'elaborazione dei dati e l'esecuzione delle milioni di operazioni richieste da una CNN, ottenendo una classificazione in pochi secondi. Inoltre, tale limitazione può essere ovviata mettendo in pratica specifiche implementazioni allo scopo di supportare ulteriori dispositivi.
I risultati descritti sembrano essere molto promettenti, sia per quanto riguarda la possibilità di incorporare modelli complessi all'interno di un semplice

smartphone, sia per consentire l'attuazione di progetti particolarmente specifici. Dopotutto, anche se molte delle funzioni presenti negli smartphone moderni siano basate sul Machine Learning, non si tratta di implementazioni facilmente realizzabili, a causa dell'elevata specificità delle poche informazioni disponibili o alla mancanza delle risorse necessarie. Per questo, uno degli obiettivi preposti è stato anche quello di contribuire allo sviluppo e al diffondersi di queste tecnologie in ambito iOS.

Sviluppi Futuri

Il lavoro svolto in questa tesi, con particolare riferimento all'applicazione iOS, si presta molto bene a miglioramenti ed espansioni.

Nello specifico, una prima area di intervento consiste nel supportare altri dispositivi, attualmente esclusi a causa dei requisiti dei Metal Performance Shaders. Si potrebbe infatti considerare l'implementazione della stessa rete neurale senza lo sfruttamento della GPU, utilizzando le Basic Neural Network Subroutines, nel caso in cui si riuscissero a realizzare tutti i filtri ed i livelli necessari non presenti nel framework. Un'ulteriore alternativa consisterebbe nella predisposizione di un server remoto a cui assegnare il compito di eseguire l'inferenza. Entrambe le soluzioni presentano alcuni pregi e difetti: nel primo caso, non si può ottenere la stessa velocità degli MPS e si è, in un certo senso, limitati ai dispositivi dell'ecosistema Apple mentre, nel secondo caso, l'utilizzo dell'applicazione sarebbe dipendente da una connessione ad internet, sebbene sia possibile il supporto di apparati più generici. Inoltre, utilizzando un server remoto, un'eventuale espansione della rete neurale potrebbe essere messa in atto senza dover aggiornare l'applicazione da parte degli utenti.

In aggiunta, il dataset alla base del *training* è prettamente generico; risulterebbe utile specializzarlo, in modo da classificare precisi medicinali, come ad esempio quelli cardiaci. Va quindi perfezionato in base al dominio in esame. Contestualmente, le immagini andrebbero acquisite con attrezzature

migliori, come attraverso l'uso del macro per poter contare su una maggiore nitidezza dei dettagli, quali i contorni e gli imprint sulle pillole.

Similmente, anche la struttura della rete neurale stessa può essere rivista ed ampliata, se necessario, al fine di individuare migliori *hyperparametri* per l'allenamento e per gestire un'eventuale quantità maggiore di dati e classi.

Infine, si potrebbe meditare di utilizzare l'applicazione per stabilire una comunicazione tra medici e pazienti, magari per supervisionare l'andamento di una terapia farmacologica, il tutto tramite l'impiego dell'applicazione da parte dei pazienti e di un sito web da parte del personale sanitario.

Appendice A

PillRecogNet in Dettaglio

A.1 Struttura Generale

Di seguito viene illustrata la struttura, livello per livello, della ConvNet realizzata in questa tesi; in particolare, si ha un'indicazione sul tipo di livello (`Conv2D` per un livello convoluzionale, `MaxPooling2D` per un livello di *max pooling*), sulla "forma" del tensore che ognuno di essi ritorna in output e sul numero di pesi associati. `Sequential` è un tipo di livello disponibile in Keras utilizzato per racchiudere un gruppo di *layer*, i quali sono mostrati nella sezione successiva.

Come si evince, tale rete è formata da circa 28 milioni di parametri ed accetta in input immagini RGB da 224×224 pixel; il numero di filtri (e quindi la profondità dei tensori) raddoppia in seguito ad ogni livello di *pooling*, partendo da 64 fino a 512 mentre `None` sta ad indicare un *placeholder* per il numero variabile di immagini da analizzare.

Tutti i livelli fino a `block5_pool` sono equivalenti alla struttura predefinita di VGG16, sebbene i parametri degli ultimi tre livelli convoluzionali siano stati aggiornati con il *fine tuning*.
In dettaglio, tutti i livelli convoluzionali hanno un *kernel* di dimensione 3×3

ed uno *stride* pari a 1 mentre, i livelli di *max pooling* sono 2×2 con uno *stride* di 2. Inoltre, tutti gli *hidden layer* applicano la non linearità ReLU classica.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 224, 224, 3)       0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
```

```
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)      2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)      0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)      2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)      2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)      2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)        0
_________________________________________________________________
sequential_1 (Sequential)    (None, 12)               12851724
=================================================================
Total params: 27,566,412
Trainable params: 19,931,148
Non-trainable params: 7,635,264
_________________________________________________________________
```

A.2 Livelli Finali Personalizzati

I livelli successivi rappresentano il contenuto di `sequential_1`, ovvero la piccola rete neurale allenata per riconoscere i 12 medicinali in esame e che sostituisce i *fully connected layers* della VGG16.
`Flatten` è un livello fornito da Keras che viene utilizzato per manipolare la forma del tensore e renderlo compatibile con il formato in input richiesto da un livello *fully connected* (chiamato `Dense` in questo caso). Inoltre, è stata utilizzata una probabilità $p = 0,5$ come parametro per il *dropout.* Il risultato è un tensore associato alle 12 classi da cui è possibile ricavare le relative probabilità e, quindi, i risultati della classificazione.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               12845568
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 12)                6156
=================================================================
Total params: 12,851,724
Trainable params: 12,851,724
Non-trainable params: 0
_________________________________________________________________
```

Appendice B

Implementazione

B.1 Python

B.1.1 Feature Extraction

```
1 from keras.preprocessing.image import ImageDataGenerator
2 from keras.utils import to_categorical
3 from keras.optimizers import SGD
4 from keras.applications import VGG16
5 import os
6 import numpy as np
7
8 import utils as u
9 import settings as s
10
11 # file paths
12 bottleneck_train_datapath = os.path.join(s.results_dir,
    "bottlenecks_train.npy")
13 bottleneck_valid_datapath = os.path.join(s.results_dir,
    "bottlenecks_validation.npy")
14 bottleneck_accuracy_plot_path = os.path.join(s.plots_dir,
    "bottleneck_accuracy.png")
15 bottleneck_loss_plot_path = os.path.join(s.plots_dir, "bottleneck_loss.png")
16
17 def save_bottlebeck_features():
18     print("Bottleneck features calculation started...")
19
20     # Load VGG16 network without top layers to act as a feature extractor
21     model = VGG16(include_top=False, weights="imagenet")
22
```

```
23     # Get features for train and validation images and save them
24     datagen = ImageDataGenerator(rescale=1. / 255,
       preprocessing_function=u.meanSubtraction)
25     generator = datagen.flow_from_directory(s.train_data_dir,
       target_size=(s.img_width, s.img_height), batch_size=s.batch_size,
       class_mode=None, shuffle=False)
26     bottleneck_features_train = model.predict_generator(generator,
       s.nb_train_samples // s.batch_size, verbose=1)
27     np.save(open(bottleneck_train_datapath, "w"), bottleneck_features_train)
28
29     generator = datagen.flow_from_directory(s.validation_data_dir,
       target_size=(s.img_width, s.img_height), batch_size=s.batch_size,
       class_mode=None, shuffle=False)
30     bottleneck_features_validation = model.predict_generator(generator,
       s.nb_validation_samples // s.batch_size, verbose=1)
31     np.save(open(bottleneck_valid_datapath, "w"),
       bottleneck_features_validation)
32
33     print("Bottlenecks saved...")
34
35 def train_top_model():
36     print("Top model training started...")
37
38     train_per_class = s.nb_train_samples // s.num_classes
39     valid_per_class = s.nb_validation_samples // s.num_classes
40
41     # Load saved features from bottlenecks
42     train_data = np.load(open(bottleneck_train_datapath))
43     train_labels = np.array([0] * train_per_class + [1] * train_per_class +
       [2] * train_per_class + [3] * train_per_class + [4] * train_per_class +
       [5] * train_per_class + [6] * train_per_class + [7] * train_per_class +
       [8] * train_per_class + [9] * train_per_class + [10] * train_per_class
       + [11] * train_per_class)
44
45     validation_data = np.load(open(bottleneck_valid_datapath))
46     validation_labels = np.array([0] * valid_per_class + [1] *
       valid_per_class + [2] * valid_per_class + [3] * valid_per_class + [4] *
       valid_per_class + [5] * valid_per_class + [6] * valid_per_class + [7] *
       valid_per_class + [8] * valid_per_class + [9] * valid_per_class + [10]
       * valid_per_class + [11] * valid_per_class)
47
48     train_labels = to_categorical(train_labels, num_classes=s.num_classes)
49     validation_labels = to_categorical(validation_labels,
       num_classes=s.num_classes)
50
51     # Create new top layers
52     model = u.obtainNewTopLayers(train_data.shape[1:], s.num_classes)
```

```
53
54     # Compile the model using Stocastic Gradient Descent and a low learning
       rate
55     optimizer = SGD(lr=1e-3, momentum=0.9)
56     model.compile(optimizer=optimizer, loss="categorical_crossentropy",
       metrics=["accuracy"])
57
58     # Start training...
59     print("Fitting...")
60     history = model.fit(train_data, train_labels, epochs=s.botEpochs,
       batch_size=s.batch_size, validation_data=(validation_data,
       validation_labels), verbose=1)
61
62     model.save_weights(s.top_model_weights_path)
63     model.save(s.top_model_model_path)
64
65     print("Model and weights saved...")
66
67     # Create graphs
68     legend = ["Training", "Validation"]
69     accData = [history.history["acc"], history.history["val_acc"]]
70     lossData = [history.history["loss"], history.history["val_loss"]]
71     u.plotGraph(accData, "Feature Extraction Accuracy", "Epoch",
       "Accuracy", legend, bottleneck_accuracy_plot_path)
72     u.plotGraph(lossData, "Feature Extraction Loss", "Epoch", "Loss",
       legend, bottleneck_loss_plot_path)
73
74 if __name__ == "__main__":
75     u.checkDirs([s.train_data_dir, s.validation_data_dir, s.test_data_dir])
76     u.createDirIfNotExisting(s.plots_dir)
77     u.createDirIfNotExisting(s.results_dir)
78     save_bottlebeck_features()
79     train_top_model()
80     print("Done!")
```

Listato B.1: Codice Python che si occupa dell'estrazione delle *feature* per l'inizializzazione della rete neurale aggiuntiva

B.1.2 Fine Tuning

```
1 from keras.applications import VGG16
2 from keras.preprocessing.image import ImageDataGenerator
3 from keras.optimizers import SGD
4 from keras.models import Model
5 from keras.callbacks import LearningRateScheduler
6 import os
7 import math
```

```
8
9  import settings as s
10 import utils as u
11
12 # File paths
13 fine_tuning_acc_plot_path = os.path.join(s.plots_dir,
       "fine_tuning_accuracy.png")
14 fine_tuning_loss_plot_path = os.path.join(s.plots_dir,
       "fine_tuning_loss.png")
15 fine_tuning_alr_plot_path = os.path.join(s.plots_dir, "fine_tuning_alr.png")
16
17
18 # Build VGG16 network
19 base_model = VGG16(weights="imagenet", include_top=False,
       input_shape=(s.img_width,s.img_height,3))
20
21 # Obtain custom top layers configuration
22 top_model = u.obtainNewTopLayers(base_model.output_shape[1:], s.num_classes)
23
24 # Load top layers weights from exported features
25 print("Loading top weights...")
26 top_model.load_weights(s.top_model_weights_path)
27
28 # Attach custom top layers to VGG16 network
29 model = Model(inputs=base_model.input, outputs=top_model(base_model.output))
30
31 # Set VGG16 layers to be non trainable so that only the last
32 # convolutional block will be fine tuned
33 for layer in model.layers[:15]:
34     layer.trainable = False
35
36 # Compile the merged network; lr = 0.0 means that no learning rate
37 # has been specified since Adaptive Learning Rate will be used
38 optimizer = SGD(lr=0.0, momentum=0.9)
39 model.compile(loss="categorical_crossentropy", optimizer=optimizer,
       metrics=["accuracy"])
40
41 # Apply image augmentation to training samples and elaborate them
42 train_datagen = ImageDataGenerator(rescale=1. / 255, shear_range=0.2,
       zoom_range=0.2, width_shift_range=0.02, height_shift_range=0.02,
       rotation_range=20, horizontal_flip=True,
       preprocessing_function=u.meanSubtraction)
43 train_generator = train_datagen.flow_from_directory(s.train_data_dir,
       target_size=(s.img_height, s.img_width), batch_size=s.batch_size,
       class_mode="categorical")
44
45 # Load validation images and elaborate them
```

```
46 valid_datagen = ImageDataGenerator(rescale=1. / 255,
       preprocessing_function=u.meanSubtraction)
47 validation_generator =
       valid_datagen.flow_from_directory(s.validation_data_dir,
       target_size=(s.img_height, s.img_width), batch_size=s.batch_size,
       class_mode="categorical")
48 
49 print("Getting validation class indices...")
50 print(validation_generator.class_indices)
51 
52 # Define Adaptive Learning Rate function to be used as a callback
53 lrs = [0.001]
54 def ALR(epoch):
55     initialLR = 0.001
56     drop = 0.5
57     epochsDrop = 17
58     newLR = initialLR * math.pow(drop, math.floor((1+epoch)/epochsDrop))
59     lrs.append(newLR)
60     print("\nCurrent LR = {:.7f}".format(newLR))
61     return newLR
62 
63 # Prepare the ALR callback
64 lrSched = LearningRateScheduler(ALR)
65 
66 # Start fine tuning...
67 print("Fitting...")
68 history = model.fit_generator(train_generator,
       steps_per_epoch=s.nb_train_samples // s.batch_size, epochs=s.ftEpochs,
       validation_data=validation_generator,
       validation_steps=s.nb_validation_samples // s.batch_size, verbose=1,
       callbacks=[lrSched])
69 
70 model.save_weights(s.fine_tuned_weights_path)
71 model.save(s.fine_tuned_model_path)
72 
73 print("Model and weights saved...")
74 print("Trying to evaluate...")
75 
76 # Evaluate fine tuned model
77 metrics = model.evaluate_generator(validation_generator,
       steps=s.nb_validation_samples // s.batch_size)
78 statsDict = dict(zip(model.metrics_names, metrics))
79 print(statsDict)
80 
81 # Generate and save fine tuning plots
82 print("Saving plots...")
83 legend = ["Training", "Validation"]
```

```
84 accData = [history.history["acc"], history.history["val_acc"]]
85 lossData = [history.history["loss"], history.history["val_loss"]]
86 u.plotGraph(accData, "Fine Tuning Accuracy", "Epoch", "Accuracy", legend,
       fine_tuning_acc_plot_path)
87 u.plotGraph(lossData, "Fine Tuning Loss", "Epoch", "Loss", legend,
       fine_tuning_loss_plot_path)
88 u.plotGraph([lrs], "Learning Rates", "Epoch", "Learning Rate", None,
       fine_tuning_alr_plot_path)
89
90 print("Done!")
```

Listato B.2: Codice Python che effettua il *fine tuning* della rete neurale aggiuntiva, al fine di migliorarne l'accuratezza

B.1.3 Valutazione della Rete Neurale

```
1 from keras.models import load_model
2 from keras.preprocessing.image import ImageDataGenerator, img_to_array,
      load_img
3 import numpy as np
4 import os
5 import glob
6
7 import settings as s
8 import utils as u
9
10 # File paths
11 log_path = os.path.join(s.results_dir, "log_test_dataset.txt")
12
13 # Check test data existence
14 u.checkDirs([s.test_data_dir])
15
16 # Load fine tuned model
17 model = load_model(s.fine_tuned_model_path)
18 print("Fine tuned model loaded...")
19
20 # Prepare test images for prediction
21 datagen = ImageDataGenerator(rescale=1./255,
       preprocessing_function=u.meanSubtraction)
22 test_generator = datagen.flow_from_directory(s.test_data_dir,
       target_size=(s.img_width, s.img_height), batch_size=s.test_batch_size,
       class_mode="categorical")
23
24 pillDictionary = test_generator.class_indices
25 print(pillDictionary)
26
27 total = 0
```

```
28 correct = 0
29
30 # Create log file for reference about predictions
31 with open(log_path, "w") as logFile:
32     logFile.write("Pill Image\tReal Class\tPredicted
    Class\n----------\t----------\t---------------\n")
33     for imagePath in glob.glob(os.path.join(s.test_data_dir, "*/*.jpg")):
34         print("\n\nTrying to classificate " + imagePath)
35
36         # Load image, preprocess it and classify it
37         image = load_img(imagePath, target_size=(s.img_width, s.img_height))
38         image = img_to_array(image)
39         image = u.meanSubtraction(image)
40         image = image / 255
41         image = np.expand_dims(image, axis=0)
42         prediction = model.predict(image)
43
44         # Get ID of the best prediction to obtain the associated class label
45         ID = prediction.argmax()
46         mapping = {v: k for k, v in pillDictionary.items()}
47         label = mapping[ID]
48
49         total += 1
50         pathComponents = imagePath.split("/")
51         if pathComponents[2] == label:
52             correct += 1
53         else:
54             # print prediction value if classification is wrong
55             print(prediction)
56
57         print("Predicted Class ID: {}, Label: {}, Real Label: {}, Correct?
    {}".format(ID, label, pathComponents[2], pathComponents[2] == label))
58         logFile.write("{}\t{}\t{}\n".format(pathComponents[3],
    pathComponents[2], label))
59
60     finalMsg = "Total images classified: {}, total correct: {}, final
    ratio: {:.2f}%".format(total, correct, correct * 100.0 / total)
61     print(finalMsg)
62     logFile.write(finalMsg)
63
64 print("Done!")
```

Listato B.3: Codice Python utilizzato per valutare l'accuratezza della rete neurale su ogni immagine del dataset di test

B.1.4 Esportazione Parametri

```
from keras.models import load_model
import os

import settings as s
import utils as u

def exportWeights():
    # Load fine tuned model with weights
    model = load_model(s.fine_tuned_model_path)
    print("Model loaded...")
    print("Getting weights...")

    W = model.get_weights()

    # Reshape and transpose are necessary to use weights with Metal
    Performance Shaders
    for i, w in enumerate(W):
        j = i // 2 + 1
        # print for info
        data = "weights" if i % 2 == 0 else "bias"
        print("Layer {}: exporting {} with shape: {}...".format(j, data,
    w.shape))

        # Handle fully connected weights
        if (j == 14 or j == 15) and i % 2 == 0:
            fc_shape = (7, 7, 512, 512) if (j == 14) else (1, 1, 512, 12)
            num = j - 13

            channel1 = fc_shape[0] * fc_shape[1] * fc_shape[2]
            channel2 = fc_shape[3]
            handledFCWeights = w.reshape(fc_shape).transpose(3, 0, 1,
    2).reshape(channel1, channel2)
            handledFCWeights.tofile(os.path.join(s.params_path,
    "fc%d_weights.bin" % num))
        # Handle fully connected bias
        elif (j == 14 or j == 15) and i % 2 == 1:
            num = j - 13
            w.tofile(os.path.join(s.params_path, "fc%d_bias.bin" % num))
        # Handle convolutional weights and bias
        else:
            if i % 2 == 0:
                w.transpose(3, 0, 1, 2).tofile(os.path.join(s.params_path,
    "conv%d_weights.bin" % j))
            else:
                w.tofile(os.path.join(s.params_path, "conv%d_bias.bin" % j))

```

```
42 if __name__ == "__main__":
43     u.createDirIfNotExisting(s.params_path)
44     exportWeights()
45     print("Done!")
```

Listato B.4: Codice Python necessario all'esportazione dei parametri della rete neurale per utilizzarli nell'applicazione iOS

B.1.5 Codice Condiviso

Impostazioni del training

```
1  import os
2
3  # Shared training settings
4  # Image size accepted by CNN
5  img_width, img_height = 224, 224
6
7  # Training settings
8  # Train and validation sample numbers MUST be divisible by batch_size
9  nb_train_samples = 528
10 nb_validation_samples = 96
11 botEpochs = 100
12 ftEpochs = 50
13 batch_size = 16
14 test_batch_size = 12
15 num_classes = 12
16
17 # folder paths
18 plots_dir = "plots"
19 results_dir = "results"
20 params_path = "params"
21 train_data_dir = "data/train"
22 validation_data_dir = "data/validation"
23 test_data_dir = "data/test"
24
25 top_model_weights_path = os.path.join(results_dir,
       "custom_layers_bottlenecks.h5")
26 top_model_model_path = os.path.join(results_dir, "bottleneck_model.h5")
27
28 fine_tuned_model_path = os.path.join(results_dir, "fine-tuned-model.h5")
29 fine_tuned_weights_path = os.path.join(results_dir, "fine-tuned-weights.h5")
```

Listato B.5: Codice Python contenente le impostazioni condivise per il *training* ad il *fine tuning* della rete neurale

Funzioni helper

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dropout, Flatten, Dense
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plot
import os
import errno

# Preprocessing function to subtract ImageNet mean RGB values from new
    images
# to keep weights coherent during transfer learning
def meanSubtraction(x):
    x = x.astype(np.float32)
    means = np.array([123.68, 116.779, 103.939],
    dtype=np.float32).reshape((1,1,3))
    x -= means
    return x

# Helper function to get new top layers configuration
def obtainNewTopLayers(input_shape, num_classes):
    model = Sequential()
    model.add(Flatten(input_shape=input_shape))
    model.add(Dense(512, activation="relu"))
    model.add(Dropout(0.5))
    model.add(Dense(num_classes, activation="softmax"))
    return model

# Helper function to plot and save graphs about the training
def plotGraph(graphData, title, xlabel, ylabel, legend, savePath):
    for (i, data) in enumerate(graphData):
        plot.plot(data)
    plot.title(title)
    plot.ylabel(ylabel)
    plot.xlabel(xlabel)
    if legend != None:
        plot.legend(legend, loc="lower right")
    plot.savefig(savePath)
    plot.close()

# Helper function to organize folder structure
def createDirIfNotExisting(path):
    try:
        os.makedirs(path)
    except OSError as e:
        if e.errno != errno.EEXIST:
```

```
45             raise
46
47 # Raise exception if training data is not found
48 def checkDirs(paths):
49     for (i, path) in enumerate(paths):
50         if not os.path.isdir(path):
51             raise IOError(errno.ENOENT, os.strerror(errno.ENOENT), path)
```

Listato B.6: Codice Python contenente alcune delle funzioni condivise tra i vari script

B.2 Swift e Metal

B.2.1 PillRecogNet

```
1  import Foundation
2  import MetalPerformanceShaders
3  import QuartzCore
4
5  private func createPoolingMax(device: MTLDevice) -> MPSCNNPoolingMax {
6    let pooling = MPSCNNPoolingMax(device: device, kernelWidth: 2,
       kernelHeight: 2, strideInPixelsX: 2, strideInPixelsY: 2)
7    pooling.offset = MPSOffset(x: 1, y: 1, z: 0)
8    return pooling
9  }
10
11 class PillConvolution: MPSCNNConvolution {
12   private let convSize:UInt = 3 * 3
13
14   required init?(coder aDecoder: NSCoder) {
15     fatalError("init(coder:) has not been implemented")
16   }
17   init(device: MTLDevice, inputDepth: UInt, outputDepth: UInt,
       parametersName: String, filter: MPSCNNNeuron?) {
18     let sizeBias = outputDepth * UInt(MemoryLayout<Float>.size)
19     let sizeWeights = inputDepth * outputDepth * convSize *
       UInt(MemoryLayout<Float>.size)
20
21     let wPath = Bundle.main.path(forResource: parametersName + "_weights",
       ofType: "bin")
22     let bPath = Bundle.main.path(forResource: parametersName + "_bias",
       ofType: "bin")
23
24     let wFileDesc = open(wPath!, O_RDONLY, S_IRUSR | S_IWUSR | S_IRGRP |
       S_IWGRP | S_IROTH | S_IWOTH)
```

```
25      let bFileDesc = open(bPath!, O_RDONLY, S_IRUSR | S_IWUSR | S_IRGRP |
        S_IWGRP | S_IROTH | S_IWOTH)
26      assert(wFileDesc != -1, "Errore nell'apertura del file \(wPath!),
        numero: \(errno)")
27      assert(bFileDesc != -1, "Errore nell'apertura del file \(bPath!),
        numero: \(errno)")
28
29      let wMem = mmap(nil, Int(sizeWeights), PROT_READ, MAP_FILE |
        MAP_SHARED, wFileDesc, 0)
30      let bMem = mmap(nil, Int(sizeBias), PROT_READ, MAP_FILE | MAP_SHARED,
        bFileDesc, 0)
31
32      let weights = UnsafePointer(wMem?.bindMemory(to: Float.self, capacity:
        Int(sizeWeights)))
33      let bias = UnsafePointer(bMem?.bindMemory(to: Float.self, capacity:
        Int(sizeBias)))
34      assert(weights != UnsafePointer<Float>.init(bitPattern: -1), "mmap
        fallita! Errore: \(errno)")
35      assert(bias != UnsafePointer<Float>.init(bitPattern: -1), "mmap
        fallita! Errore: \(errno)")
36
37      let convDescriptor = MPSCNNConvolutionDescriptor(kernelWidth: 3,
        kernelHeight: 3, inputFeatureChannels: Int(inputDepth),
        outputFeatureChannels: Int(outputDepth), neuronFilter: filter)
38      convDescriptor.strideInPixelsX = 1
39      convDescriptor.strideInPixelsY = 1
40
41      super.init(device: device, convolutionDescriptor: convDescriptor,
        kernelWeights: weights!, biasTerms: bias!, flags: .none)
42      self.edgeMode = .zero
43
44      assert(munmap(wMem, Int(sizeWeights)) == 0, "munmap fallita! Errore:
        \(errno)")
45      assert(munmap(bMem, Int(sizeBias)) == 0, "munmap fallita! Errore:
        \(errno)")
46      close(wFileDesc)
47      close(bFileDesc)
48    }
49  }
50  class PillFullyConnected: MPSCNNFullyConnected {
51    required init?(coder aDecoder: NSCoder) {
52      fatalError("init(coder:) has not been implemented")
53    }
54    init(device: MTLDevice, kernelSize: UInt, inputDepth: UInt, outputDepth:
        UInt, parametersName: String, filter: MPSCNNNeuron?) {
55      let sizeBias = outputDepth * UInt(MemoryLayout<Float>.size)
56      let sizeWeights = inputDepth * kernelSize * kernelSize * outputDepth *
```

```
        UInt(MemoryLayout<Float>.size)
57
58      let wPath = Bundle.main.path(forResource: parametersName + "_weights",
        ofType: "bin")
59      let bPath = Bundle.main.path(forResource: parametersName + "_bias",
        ofType: "bin")
60
61      let wFileDesc = open(wPath!, O_RDONLY, S_IRUSR | S_IWUSR | S_IRGRP |
        S_IWGRP | S_IROTH | S_IWOTH)
62      let bFileDesc = open(bPath!, O_RDONLY, S_IRUSR | S_IWUSR | S_IRGRP |
        S_IWGRP | S_IROTH | S_IWOTH)
63      assert(wFileDesc != -1, "Errore nell'apertura del file \(wPath!),
        numero: \(errno)")
64      assert(bFileDesc != -1, "Errore nell'apertura del file \(bPath!),
        numero: \(errno)")
65
66      let wMem = mmap(nil, Int(sizeWeights), PROT_READ, MAP_FILE |
        MAP_SHARED, wFileDesc, 0)
67      let bMem = mmap(nil, Int(sizeBias), PROT_READ, MAP_FILE | MAP_SHARED,
        bFileDesc, 0)
68
69      let weights = UnsafePointer(wMem?.bindMemory(to: Float.self, capacity:
        Int(sizeWeights)))
70      let bias = UnsafePointer(bMem?.bindMemory(to: Float.self, capacity:
        Int(sizeBias)))
71      assert(weights != UnsafePointer<Float>.init(bitPattern: -1), "mmap
        fallita! Errore: \(errno)")
72      assert(bias != UnsafePointer<Float>.init(bitPattern: -1), "mmap
        fallita! Errore: \(errno)")
73
74      let fcDescriptor = MPSCNNConvolutionDescriptor(kernelWidth:
        Int(kernelSize), kernelHeight: Int(kernelSize), inputFeatureChannels:
        Int(inputDepth), outputFeatureChannels: Int(outputDepth), neuronFilter:
        filter)
75
76      super.init(device: device, convolutionDescriptor: fcDescriptor,
        kernelWeights: weights!, biasTerms: bias!, flags: .none)
77
78      assert(munmap(wMem, Int(sizeWeights)) == 0, "munmap fallita! Errore:
        \(errno)")
79      assert(munmap(bMem, Int(sizeBias)) == 0, "munmap fallita! Errore:
        \(errno)")
80      close(wFileDesc)
81      close(bFileDesc)
82    }
83  }
84  public class PillRecogNet {
```

```
85  let device: MTLDevice
86  let commandQueue: MTLCommandQueue
87  let rgbPipeline: MTLComputePipelineState
88  let outputImage: MPSImage
89
90  let lanczos: MPSImageLanczosScale
91  let relu: MPSCNNNeuronReLU
92  let softmax: MPSCNNSoftMax
93
94  let block1_conv1, block1_conv2: PillConvolution
95  let block2_conv1, block2_conv2: PillConvolution
96  let block3_conv1, block3_conv2, block3_conv3: PillConvolution
97  let block4_conv1, block4_conv2, block4_conv3: PillConvolution
98  let block5_conv1, block5_conv2, block5_conv3: PillConvolution
99
100 let block1_pool, block2_pool, block3_pool, block4_pool, block5_pool:
      MPSCNNPoolingMax
101 let dense_1, dense_2: PillFullyConnected
102
103 let inputImgDesc = MPSImageDescriptor(channelFormat: .float16, width:
      224, height: 224, featureChannels: 3)
104 let conv_block1ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 224, height: 224, featureChannels: 64)
105 let pool_block1ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 112, height: 112, featureChannels: 64)
106 let conv_block2ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 112, height: 112, featureChannels: 128)
107 let pool_block2ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 56, height: 56, featureChannels: 128)
108 let conv_block3ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 56, height: 56, featureChannels: 256)
109 let pool_block3ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 28, height: 28, featureChannels: 256)
110 let conv_block4ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 28, height: 28, featureChannels: 512)
111 let pool_block4ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 14, height: 14, featureChannels: 512)
112 let conv_block5ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 14, height: 14, featureChannels: 512)
113 let pool_block5ImgDesc = MPSImageDescriptor(channelFormat: .float16,
      width: 7, height: 7, featureChannels: 512)
114 let denseImgDesc = MPSImageDescriptor(channelFormat: .float16, width: 1,
      height: 1, featureChannels: 512)
115 let outputImgDesc = MPSImageDescriptor(channelFormat: .float16, width: 1,
      height: 1, featureChannels: 12)
116
117 var imageDescriptors:[MPSImageDescriptor] {
```

```
118     get {
119       return [inputImgDesc, conv_block1ImgDesc, pool_block1ImgDesc,
        conv_block2ImgDesc, pool_block2ImgDesc, conv_block3ImgDesc,
        pool_block3ImgDesc, conv_block4ImgDesc, pool_block4ImgDesc,
        conv_block5ImgDesc, pool_block5ImgDesc, denseImgDesc, outputImgDesc]
120     }
121   }
122   let labelManager = PillLabelManager()
123
124   init(device: MTLDevice) {
125     self.device = device
126     commandQueue = device.makeCommandQueue()!
127     outputImage = MPSImage(device: device, imageDescriptor: outputImgDesc)
128     lanczos = MPSImageLanczosScale(device: device)
129     relu = MPSCNNNeuronReLU(device: device, a: 0)
130     softmax = MPSCNNSoftMax(device: device)
131
132     do {
133       let customFunctionsLibrary = device.makeDefaultLibrary()!
134       let vggPreprocessing = customFunctionsLibrary.makeFunction(name:
        "removeRGBMean")
135       rgbPipeline = try device.makeComputePipelineState(function:
        vggPreprocessing!)
136     } catch {
137       fatalError("Errore durante l'inizializzazione del kernel per il
        preprocessing!")
138     }
139     block1_conv1 = PillConvolution(device: device, inputDepth: 3,
        outputDepth: 64, parametersName: "conv1", filter: relu)
140     block1_conv2 = PillConvolution(device: device, inputDepth: 64,
        outputDepth: 64, parametersName: "conv2", filter: relu)
141     block1_pool = createPoolingMax(device: device)
142
143     block2_conv1 = PillConvolution(device: device, inputDepth: 64,
        outputDepth: 128, parametersName: "conv3", filter: relu)
144     block2_conv2 = PillConvolution(device: device, inputDepth: 128,
        outputDepth: 128, parametersName: "conv4", filter: relu)
145     block2_pool = createPoolingMax(device: device)
146
147     block3_conv1 = PillConvolution(device: device, inputDepth: 128,
        outputDepth: 256, parametersName: "conv5", filter: relu)
148     block3_conv2 = PillConvolution(device: device, inputDepth: 256,
        outputDepth: 256, parametersName: "conv6", filter: relu)
149     block3_conv3 = PillConvolution(device: device, inputDepth: 256,
        outputDepth: 256, parametersName: "conv7", filter: relu)
150     block3_pool = createPoolingMax(device: device)
151
```

```
152     block4_conv1 = PillConvolution(device: device, inputDepth: 256,
        outputDepth: 512, parametersName: "conv8", filter: relu)
153     block4_conv2 = PillConvolution(device: device, inputDepth: 512,
        outputDepth: 512, parametersName: "conv9", filter: relu)
154     block4_conv3 = PillConvolution(device: device, inputDepth: 512,
        outputDepth: 512, parametersName: "conv10", filter: relu)
155     block4_pool = createPoolingMax(device: device)
156
157     block5_conv1 = PillConvolution(device: device, inputDepth: 512,
        outputDepth: 512, parametersName: "conv11", filter: relu)
158     block5_conv2 = PillConvolution(device: device, inputDepth: 512,
        outputDepth: 512, parametersName: "conv12", filter: relu)
159     block5_conv3 = PillConvolution(device: device, inputDepth: 512,
        outputDepth: 512, parametersName: "conv13", filter: relu)
160     block5_pool = createPoolingMax(device: device)
161
162     dense_1 = PillFullyConnected(device: device, kernelSize: 7, inputDepth:
        512, outputDepth: 512, parametersName: "fc1", filter: relu)
163     dense_2 = PillFullyConnected(device: device, kernelSize: 1, inputDepth:
        512, outputDepth: 12, parametersName: "fc2", filter: nil)
164
165     for desc in self.imageDescriptors {
166       desc.storageMode = .private
167     }
168   }
169   func classify(pill inputImage: MPSImage) -> [PillMatch] {
170     autoreleasepool {
171       if let comBuf = commandQueue.makeCommandBuffer() {
172         MPSTemporaryImage.prefetchStorage(with: comBuf,
        imageDescriptorList: self.imageDescriptors)
173
174         let scaledImage = MPSTemporaryImage(commandBuffer: comBuf,
        imageDescriptor: inputImgDesc)
175         lanczos.encode(commandBuffer: comBuf, sourceTexture:
        inputImage.texture, destinationTexture: scaledImage.texture)
176
177         let preprocessedImage = MPSTemporaryImage(commandBuffer: comBuf,
        imageDescriptor: inputImgDesc)
178
179         let encoder = comBuf.makeComputeCommandEncoder()
180         encoder?.setComputePipelineState(rgbPipeline)
181         encoder?.setTexture(scaledImage.texture, index: 0)
182         encoder?.setTexture(preprocessedImage.texture, index: 1)
183
184         let threadsPerGroups = MTLSizeMake(8, 8, 1)
185         let threadGroups = MTLSizeMake(preprocessedImage.texture.width /
        threadsPerGroups.width, preprocessedImage.texture.height /
```

```
threadsPerGroups.height, 1)
186        encoder?.dispatchThreadgroups(threadGroups, threadsPerThreadgroup:
    threadsPerGroups)
187        encoder?.endEncoding()
188        scaledImage.readCount = 0
189
190        let b1c1Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block1ImgDesc)
191        block1_conv1.encode(commandBuffer: comBuf, sourceImage:
    preprocessedImage, destinationImage: b1c1Img)
192        preprocessedImage.readCount = 0
193
194        let b1c2Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block1ImgDesc)
195        block1_conv2.encode(commandBuffer: comBuf, sourceImage: b1c1Img,
    destinationImage: b1c2Img)
196        b1c1Img.readCount = 0
197
198        let b1poolImg = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: pool_block1ImgDesc)
199        block1_pool.encode(commandBuffer: comBuf, sourceImage: b1c2Img,
    destinationImage: b1poolImg)
200        b1c2Img.readCount = 0
201
202        let b2c1Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block2ImgDesc)
203        block2_conv1.encode(commandBuffer: comBuf, sourceImage: b1poolImg,
    destinationImage: b2c1Img)
204        b1poolImg.readCount = 0
205
206        let b2c2Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block2ImgDesc)
207        block2_conv2.encode(commandBuffer: comBuf, sourceImage: b2c1Img,
    destinationImage: b2c2Img)
208        b2c1Img.readCount = 0
209
210        let b2poolImg = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: pool_block2ImgDesc)
211        block2_pool.encode(commandBuffer: comBuf, sourceImage: b2c2Img,
    destinationImage: b2poolImg)
212        b2c2Img.readCount = 0
213
214        let b3c1Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block3ImgDesc)
215        block3_conv1.encode(commandBuffer: comBuf, sourceImage: b2poolImg,
    destinationImage: b3c1Img)
216        b2poolImg.readCount = 0
```

```
217
218         let b3c2Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block3ImgDesc)
219         block3_conv2.encode(commandBuffer: comBuf, sourceImage: b3c1Img,
    destinationImage: b3c2Img)
220         b3c1Img.readCount = 0
221
222         let b3c3Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block3ImgDesc)
223         block3_conv3.encode(commandBuffer: comBuf, sourceImage: b3c2Img,
    destinationImage: b3c3Img)
224         b3c2Img.readCount = 0
225
226         let b3poolImg = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: pool_block3ImgDesc)
227         block3_pool.encode(commandBuffer: comBuf, sourceImage: b3c3Img,
    destinationImage: b3poolImg)
228         b3c3Img.readCount = 0
229
230         let b4c1Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block4ImgDesc)
231         block4_conv1.encode(commandBuffer: comBuf, sourceImage: b3poolImg,
    destinationImage: b4c1Img)
232         b3poolImg.readCount = 0
233
234         let b4c2Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block4ImgDesc)
235         block4_conv2.encode(commandBuffer: comBuf, sourceImage: b4c1Img,
    destinationImage: b4c2Img)
236         b4c1Img.readCount = 0
237
238         let b4c3Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block4ImgDesc)
239         block4_conv3.encode(commandBuffer: comBuf, sourceImage: b4c2Img,
    destinationImage: b4c3Img)
240         b4c2Img.readCount = 0
241
242         let b4poolImg = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: pool_block4ImgDesc)
243         block4_pool.encode(commandBuffer: comBuf, sourceImage: b4c3Img,
    destinationImage: b4poolImg)
244         b4c3Img.readCount = 0
245
246         let b5c1Img = MPSTemporaryImage(commandBuffer: comBuf,
    imageDescriptor: conv_block5ImgDesc)
247         block5_conv1.encode(commandBuffer: comBuf, sourceImage: b4poolImg,
    destinationImage: b5c1Img)
```

```
248         b4poolImg.readCount = 0
249
250         let b5c2Img = MPSTemporaryImage(commandBuffer: comBuf,
        imageDescriptor: conv_block5ImgDesc)
251         block5_conv2.encode(commandBuffer: comBuf, sourceImage: b5c1Img,
        destinationImage: b5c2Img)
252         b5c1Img.readCount = 0
253
254         let b5c3Img = MPSTemporaryImage(commandBuffer: comBuf,
        imageDescriptor: conv_block5ImgDesc)
255         block5_conv3.encode(commandBuffer: comBuf, sourceImage: b5c2Img,
        destinationImage: b5c3Img)
256         b5c2Img.readCount = 0
257
258         let b5poolImg = MPSTemporaryImage(commandBuffer: comBuf,
        imageDescriptor: pool_block5ImgDesc)
259         block5_pool.encode(commandBuffer: comBuf, sourceImage: b5c3Img,
        destinationImage: b5poolImg)
260         b5c3Img.readCount = 0
261
262         let dense1Img = MPSTemporaryImage(commandBuffer: comBuf,
        imageDescriptor: denseImgDesc)
263         dense_1.encode(commandBuffer: comBuf, sourceImage: b5poolImg,
        destinationImage: dense1Img)
264         b5poolImg.readCount = 0
265
266         let dense2Img = MPSTemporaryImage(commandBuffer: comBuf,
        imageDescriptor: outputImgDesc)
267         dense_2.encode(commandBuffer: comBuf, sourceImage: dense1Img,
        destinationImage: dense2Img)
268         dense1Img.readCount = 0
269
270         softmax.encode(commandBuffer: comBuf, sourceImage: dense2Img,
        destinationImage: outputImage)
271         dense2Img.readCount = 0
272
273         comBuf.commit()
274         comBuf.waitUntilCompleted()
275       }
276     }
277     return self.labelManager.best5Matches(probabilities:
        self.outputImage.getFloatValues())
278   }
279 }
```

Listato B.7: Implementazione dell’intera rete neurale in Swift utilizzando i Metal Performance Shaders

B.2.2 PillLabelManager

```
import Foundation

public typealias PillMatch = (label: String, probability: Float)
public class PillLabelManager {
  private static let labelCount = 12
  private var labels = [String](repeating: "", count: labelCount)

  init() {
    if let labelFile = Bundle.main.path(forResource: "pillLabels", ofType:
    "txt") {
      do {
        let content = try String(contentsOfFile: labelFile, encoding: .utf8)
        let rows = content.components(separatedBy: NSCharacterSet.newlines)

        for (i, row) in rows.enumerated() {
          if i < PillLabelManager.labelCount {
            self.labels[i] = row.components(separatedBy: "|")[1]
          }
        }
      } catch {
        fatalError("Errore durante il caricamento delle label!")
      }
    }
  }
  func best5Matches(probabilities: [Float]) -> [PillMatch] {
    precondition(probabilities.count == PillLabelManager.labelCount)

    let zippedSequence = zip(0...PillLabelManager.labelCount, probabilities)
    let sorted = zippedSequence.sorted { (a: (index: Int, probability:
    Float), b: (index: Int, probability: Float)) -> Bool in
      a.probability > b.probability
    }
    let best5 = sorted.prefix(through: 4)
    let result = best5.map { (elem: (index:Int, probability: Float)) ->
    PillMatch in
      (self.labels[elem.index], elem.probability)
    }
    return result
  }
}
```

Listato B.8: Codice Swift per gestire le classi riconosciute dalla rete neurale, in modo da separarle dalla sua struttura

B.2.3 Preprocessing Kernel

```
#include <metal_stdlib>
using namespace metal;

kernel void removeRGBMean(texture2d <half, access::read> inputTexture
    [[texture(0)]], texture2d <half, access::write> outputTexture
    [[texture(1)]], uint2 gid [[thread_position_in_grid]]) {

    const auto means = half4(123.68h, 116.78h, 103.94h, 0.0h);
    const auto inputColors = (half4(inputTexture.read(gid)) * 255.0h -
    means);
    const auto scaled = inputColors / 255.0h;
    outputTexture.write(half4(scaled.x, scaled.y, scaled.z, 0.0h), gid);
}
```

Listato B.9: Filtro personalizzato scritto in Metal Shading Language per eseguire il *preprocessing* delle immagini, sottraendo i valori R, G e B medi

Il codice Swift e Metal mostrato riguarda solo l'implementazione vera e propria della rete neurale; l'intero progetto dell'applicazione è disponibile online in maniera *open source*[1].

[1] https://github.com/matteodelv/PillRecogNet-iOS

Bibliografia

[1] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers" in IBM Journal of Research and Development, vol. 3, no. 3, pp. 210-229, July 1959.

[2] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov 1998.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. "ImageNet classification with deep convolutional neural networks" in Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12), F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 1. Curran Associates Inc., USA, 1097-1105.

[4] ImageNet LSVRC 2010. `http://image-net.org/challenges/LSVRC/2010/index`.

[5] Glorot, X., Bordes, A. and Bengio, Y.. (2011). "Deep Sparse Rectifier Neural Networks". Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, in PMLR 15:315-323.

[6] ImageNet. `http://www.image-net.org/index`.

[7] J. Deng, W. Dong, R. Socher, L. J. Li, Kai Li and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009 IEEE Conference on

Computer Vision and Pattern Recognition, Miami, FL, 2009, pp. 248-255.

[8] George A. Miller (1995). “WordNet: A Lexical Database for English”. Communications of the ACM Vol. 38, No. 11: 39-41.

[9] Olga Russakovsky*, Jia Deng*, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. (* = equal contribution) “ImageNet Large Scale Visual Recognition Challenge”. IJCV, 2015.

[10] ImageNet LSVRC. `http://www.image-net.org/challenges/LSVRC/`.

[11] C. Szegedy et al., “Going deeper with convolutions,” 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 1-9.

[12] M. Lin, Q. Chen, and S. Yan. “Network in network”. CoRR, abs/1312.4400, 2013.

[13] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, “Rethinking the Inception Architecture for Computer Vision,” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, 2016, pp. 2818-2826.

[14] Sergey Ioffe and Christian Szegedy. 2015. “Batch normalization: accelerating deep network training by reducing internal covariate shift”. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML’15), Francis Bach and David Blei (Eds.), Vol. 37. JMLR.org 448-456.

[15] K. Simonyan and A. Zisserman. “Very deep convolutional networks for large-scale image recognition”. arXiv preprint arXiv:1409.1556, 2014.

[16] S. C. Wong, A. Gatt, V. Stamatescu and M. D. McDonnell, "Understanding Data Augmentation for Classification: When to Warp?" 2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA), Gold Coast, QLD, 2016, pp. 1-6.

[17] P. Y. Simard, D. Steinkraus and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings., 2003, pp. 958-963.

[18] J. Wang and L. Perez, "The Effectiveness of Data Augmentation in Image Classification using Deep Learning". 2017 Stanford CS231N.

[19] TensorFlow. `https://www.tensorflow.org`.

[20] Theano. `http://www.deeplearning.net/software/theano/`.

[21] Caffe. `http://caffe.berkeleyvision.org`.

[22] Keras. `https://keras.io`.

[23] M. Zeiler and R. Fergus. "Visualizing and Understanding Convolutional Networks". arXiv:1311.2901, 2013.

[24] S. J. Pan and Q. Yang, "A Survey on Transfer Learning", in IEEE Transactions on Knowledge and Data Engineering, vol. 22, no. 10, pp. 1345-1359, Oct. 2010.

[25] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. "How transferable are features in deep neural networks?". In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14), Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (Eds.), Vol. 2. MIT Press, Cambridge, MA, USA, 3320-3328.

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. “Dropout: a simple way to prevent neural networks from overfitting”. J. Mach. Learn. Res. 15, 1 (January 2014), 1929-1958.

[27] Basic Neural Network Subroutines - Apple Developer Documentation. `https://developer.apple.com/documentation/accelerate/bnns`.

[28] Metal Performance Shaders - Apple Developer Documentation. `https://developer.apple.com/documentation/metalperformanceshaders`.

[29] Metal - Apple Developer Documentation. `https://developer.apple.com/documentation/metal`.

[30] Metal Shading Language Specifications - Apple Developer Documentation. `https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf`.

[31] CoreML - Apple Developer Documentation. `https://developer.apple.com/documentation/coreml`.

[32] Vision - Apple Developer Documentation. `https://developer.apple.com/documentation/vision`.

[33] NumPy. `http://www.numpy.org`.