



**UNIVERSITÀ DEGLI STUDI “ROMA TRE”**  
Dipartimento di Matematica e Fisica  
Corso di Laurea Magistrale in Scienze Computazionali  
Tesi di Laurea Magistrale

# **Ricerca della topologia ottimale di un sistema di deep learning per identificazioni di oggetti architettonici**

## **Candidato**

Désirée Adiutori

## **Relatore**

Prof. Luciano Teresi

## **Correlatore**

Prof. Roberto D'Autilia

Anno Accademico 2017/2018

# Indice

|  |           |
|--|-----------|
| Introduzione . . . . .                           | 3         |
| <b>1 Algoritmi di apprendimento</b>              | <b>4</b>  |
| 1.1 Costruzione di un algoritmo . . . . .        | 4         |
| 1.2 Apprendimento supervisionato . . . . .       | 8         |
| 1.2.1 Regressione . . . . .                      | 8         |
| 1.2.2 Classificazione . . . . .                  | 8         |
| 1.3 Apprendimento non supervisionato . . . . .   | 9         |
| 1.4 Apprendimento per rinforzo . . . . .         | 9         |
| <b>2 Reti neurali artificiali</b>                | <b>10</b> |
| 2.1 Funzioni di attivazione . . . . .            | 12        |
| 2.2 Addestramento di una rete . . . . .          | 12        |
| 2.2.1 Algoritmo di back propagation . . . . .    | 13        |
| 2.3 Reti Deep Feed-forward . . . . .             | 17        |
| 2.4 ImageNet . . . . .                           | 18        |
| 2.5 Reti Neurali Convoluzionali - CNNs . . . . . | 18        |
| 2.5.1 Struttura di una CNNs . . . . .            | 20        |
| <b>3 Shazarch</b>                                | <b>23</b> |
| 3.1 MobileNet . . . . .                          | 24        |
| 3.2 TensorFlow . . . . .                         | 26        |
| 3.2.1 Shazarch . . . . .                         | 27        |
| 3.3 AndroidStudio . . . . .                      | 30        |
| 3.3.1 Shazarch . . . . .                         | 32        |
| 3.4 Conclusioni . . . . .                        | 33        |
| <b>Bibliografia</b>                              | <b>36</b> |

## Introduzione

Dall'invenzione dei computer, l'uomo fa sempre più affidamento sulle macchine per risolvere problemi complessi di calcolo. Con l'aumentare delle prestazioni dei computer, man mano si sono sviluppati algoritmi di calcolo sempre più efficienti. Nel 1959 l'ingegnere del MIT, Arthur Samuel coniò il termine “*machine learning*”, descrivendo l'apprendimento automatico come un “campo di studio che dà ai computer la possibilità di apprendere senza essere programmati esplicitamente per farlo” [1].

Definiamo l'apprendimento automatico come un insieme di metodi in grado di rilevare automaticamente i parametri dei modelli tramite dei dati e quindi utilizzare i modelli identificati per prevedere i dati futuri o per eseguire altri tipi di processi decisionali in condizioni di incertezza.

Il “*deep learning*” è un tipo particolare di machine learning, che riguarda l'emulazione di come gli esseri umani apprendono. Esso affronta i problemi del machine learning, rappresentando il mondo come una gerarchia di concetti annidati: ogni concetto è definito in relazione a concetti più semplici e le rappresentazioni astratte vengono calcolate in termini di concetti meno astratti. Il Deep Learning implica l'utilizzo di reti neurali artificiali (*deep artificial neural networks*) algoritmi e sistemi computazionali, ispirati al cervello umano, per affrontare i problemi del Machine Learning.

Questa tesi si focalizza su un problema particolare di machine learning: la Classificazione (*Classification*) in particolar modo di immagini. L'obiettivo principale è trovare un'architettura ottimale per l'algoritmo che identifica le immagini di oggetti architettonici, inserendo tra i parametri anche le coordinate geofisiche dell'oggetto.

Nel primo capitolo si descrive cosa sono e come sono strutturati gli algoritmi di apprendimento. Nel secondo si introduce il concetto di rete neurale, ponendo particolare attenzione sulle reti neurali di tipo convoluzionale. Nel terzo si descrive il linguaggio di programmazione TensorFlow di Python; in particolar modo viene mostrato il codice utilizzato per la classificazione di oggetti architettonici e quello per la realizzazione di un App, per dispositivo Android, che lo implementi (prevalentemente codice Java). Infine nell'ultima parte vengono mostrati i risultati ottenuti.

# Capitolo 1

## Algoritmi di apprendimento

Gli algoritmi di machine learning sono solitamente divisi in tre tipi principali:

- Supervised learning (apprendimento supervisionato)
- Unsupervised learning (apprendimento non supervisionato)
- Reinforcement learning (apprendimento per rinforzo)

La scelta dell'algoritmo da utilizzare dipende dal tipo di dati di cui si dispone. Ma la scelta finale va fatta solo esclusivamente dopo aver testato l'algoritmo, e si sceglie in base a quello più performante: un insieme di ipotesi che funziona bene in un dominio, potrebbe funzionare male in un altro.

**Teorema del No Free Lunch [3, Wolper, 1996]**

*Non esiste una definizione universale di algoritmo “migliore”.*

### 1.1 Costruzione di un algoritmo

Per costruire un algoritmo di apprendimento bisogna avere:

- processi(*task*): compiti che l'algoritmo deve eseguire;
- misuratori di rendiment: rilevatori delle caratteristiche dei processi;
- esperienze: quantità di dati dal quale imparare.

I processi di apprendimento automatico descrivono come il sistema dovrebbe elaborare un esempio.

**Definizione 1** *Un esempio è una raccolta di caratteristiche che sono state misurate quantitativamente da alcuni oggetti o eventi elaborati.*

Di solito, un esempio viene rappresentato da un vettore  $x \in \mathbb{R}^n$ , dove ogni elemento  $x_i$  rappresenta una caratteristica.

Ad esempio, se si considera un fiore: le caratteristiche che lo descrivono sono la lunghezza e la larghezza dei suoi petali e il colore. Quindi in questo caso la metrica usata è la distanza euclidea tra le due estremità del petalo: se si considera  $x = (x_1, x_2, x_3)$  il fiore con queste 3 caratteristiche, si ha che:

$$x_1 = d(h_{min}, h_{max}) \quad x_2 = d(b_{min}, b_{max}) \quad x_3 = \text{stringa}$$

dove  $h_{min}$  e  $h_{max}$  rappresentano i due punti relativi alle estremità del petalo e  $d : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$  t.c  $d(x, y) = \sqrt{(x - y)^2} = |x - y|$ .

Dato un processo si cerca di capire quale sia la caratteristica principale, sulla quale si deve misurare il suo rendimento. Infine, dobbiamo dare all'algoritmo un'esperienza sulla quale apprendere, che è quella che lo classificherà in uno dei tre tipi principali. Questa esperienza l'apprende dai *dataset*: una collezione di esempi.

I dataset possono essere di vari tipi:

- di addestramento (*training set*)
- di prova (*test set*)
- di validazione (*validation set*)

L' **insieme di addestramento** è una parte dell'insieme di dati che vengono utilizzati per addestrare un sistema di apprendimento supervisionato.

Da questo insieme, l'algoritmo deve costruire una funzione che capisca, dai parametri, quali caratteristiche descrivono le varie categorie.

L' **insieme di prova** è un insieme di dati che, con l'insieme di addestramento, forma una partizione del dataset di partenza. Questi nuovi dati vengono utilizzati per valutare l'apprendimento dell'algoritmo "addestrato".

L' **insieme di validazione** è usato in maniera analoga all'insieme di prova, ma dei dati inseriti per testare l'algoritmo già si conosce la risposta (una parte di essi può far parte dell'insieme di addestramento) e da questa si valuta se l'output ottenuto è ottimale o meno.

Questi tre insiemi possono coesistere e la scelta delle loro cardinalità non è universale: dipende dal tipo di problema che viene affrontato.

Vediamo ora come valutare l'efficienza di un algoritmo:

**Definizione 2** *L'errore di allenamento* (*training error*) è una misura di errore che si può calcolare sul set di allenamento. Indica quanto l'algoritmo sta apprendendo.

**Definizione 3** La **generalizzazione** è la capacità di un algoritmo di essere ottimale in seguito ad un input proveniente dall'insieme di prova.

**Definizione 4** L'**errore di generalizzazione** (generalization error) è una misura di errore che si può calcolare sull'insieme di prova. Verifica se l'algoritmo ha imparato o solo memorizzato. Esso viene detto anche errore di test (test error).

Si ipotizza che tutti gli esempi siano eventi indipendenti e che tutti gli insiemi, in cui si partiziona l'insieme di dati, hanno la stessa distribuzione di probabilità uniforme.

**Definizione 5** Una **funzione di perdita** (loss function)

$$L(y, \hat{y}) : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$$

è una funzione che misura la distanza (o l'errore) tra i valori di output previsto  $\hat{y}$  e i valori effettivi  $y$ .

Si possono usare varie misure, ad esempio l'errore quadratico medio(MSE):

$$L(y, \hat{y})_{train} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_{(train)} - y_{(train)})_i^2 \quad (1.1)$$

La funzione di predizione dipenderà da dei parametri, rappresentati da un vettore  $w$ , lo scopo è di minimizzare l'errore di allenamento variando  $w$ . In base al tipo di apprendimento e al problema da affrontare, verranno usati vari algoritmi per risolvere problemi di minimizzazione libera. Gli algoritmi che risolverono il problema:

$$f(x^*) = \min_{x \in \mathbb{R}^n} f(x), \quad f \in C^2(\mathbb{R}^n, \mathbb{R}) \quad (1.2)$$

Per risolvere problemi di questo tipo, spesso viene utilizzato il metodo di discesa del gradiente [4].

Minimizzare l'errore di allenamento non necessariamente comporta l'ottimizzazione di apprendimento dell'algoritmo, potrebbe verificarsi il fenomeno di adattamento insufficiente (*underfitting*) ovvero non si hanno abbastanza dati per creare un modello di predizione accurato. Bisogna quindi valutare anche altri fattori: analizzare l'insieme di prova.

Ricordandoci dell'eq.1.1 calcoliamo l'errore di prova:

$$L(y, \hat{y})_{test} = \frac{1}{n} \sum_{i=1}^n (\hat{y}_{(test)} - y_{(test)})_i^2 \quad (1.3)$$

Vorremmo che, con i parametri trovati per minimizzare l'errore di allenamento, anche questo errore sia minimo (l'ottimalità è 0). Ma come detto in precedenza non sempre questo accade, vorremmo quindi che il divario tra i due errori sia minimo. In caso contrario si verifica il fenomeno di adattamento eccessivo (*overfitting*) del modello all'insieme di dati che descrive, tramite un eccessivo numero di parametri. Il modello quindi non sarà generalizzabile ad un nuovo insieme di dati.

Consideriamo il valore atteso dell'errore di prova, calcolato prendendo una coppia di punti  $(X, Y)$  dall'insieme di prova:

$$\mathbb{E}[L(y, \hat{y})_{test}] = \mathbb{E}[(Y - \hat{y}(X))^2] \quad (1.4)$$

e definiamo la funzione dell'output effettivo come:

$$y(X) = \mathbb{E}(Y|X)$$

la quale avrà sicuramente un errore, dovuto a qualche interferenza, che chiameremo: distorsione stimata (*estimation bias*).

Ma con diversi insiemi di allenamento, possiamo costruire diverse funzioni  $\hat{y}$ , e anche questo è un'altra fonte di errore: la varianza stimata (*estimation variance*). Possiamo quindi scrivere l'output come:

$$Y = y(X) + \epsilon$$

con  $\epsilon$  variabile aleatoria indipendente da  $X$ , con  $X$  distribuzione normale tale che:  $\mathbb{E}[X] = 0$  e  $Var(X) = \sigma^2$ .

Possiamo quindi riscrivere l'equazione 1.4 come:

$$\begin{aligned} \mathbb{E}[L(y, \hat{y})_{test}] &= \mathbb{E}[(Y - \hat{y}(X))^2 | X = x] \\ &= \mathbb{E}[(Y - y(x))^2 | X = x] + \mathbb{E}[(y(x) - \hat{y}(x))^2 | X = x] \\ &= \sigma^2 + \mathbb{E}[(y(x) - \hat{y}(x))^2] \end{aligned} \quad (1.5)$$

dove  $\sigma^2$  è chiamato errore Bayes e

$$\begin{aligned} \mathbb{E}[(y(x) - \hat{y}(x))^2] &= (\mathbb{E}[\hat{y}(x)] - y(x))^2 + \mathbb{E}[(\hat{y}(x) - \mathbb{E}[\hat{y}(x)])^2] \\ &= Bias(\hat{y}(x))^2 + Var(\hat{y}(x)) \end{aligned} \quad (1.6)$$

Si ottiene così il compromesso distorsione-varianza (*bias-variance tradeoff*):

$$\mathbb{E}[L(y, \hat{y})_{test}] = \sigma^2 + Bias(\hat{y}(x))^2 + Var(\hat{y}(x)) \quad (1.7)$$

Se la distorsione ha valori alti e la varianza bassi avremo un fenomeno di adattamento insufficiente, mentre se la distorsione ha valori bassi e la varianza alti avremo un adattamento eccessivo.[5]

Un modo per equilibrare questo compromesso è usare la convalida incrociata (*Cross-Validation*), che consiste nel ripetere l'addestramento e il test dell'algoritmo ogni volta su sottoinsiemi scelti in maniera casuale.

## 1.2 Apprendimento supervisionato

Gli algoritmi di apprendimento supervisionato vengono utilizzati per risolvere problemi di classificazione e di regressione.

Si parla di apprendimento supervisionato quando il dataset che si utilizza contiene delle variabili, una delle quali è un'etichetta.

Dato un vettore di input  $x = (x_1, \dots, x_n)$  ogni  $x_i$  è un vettore d-dimensionale di numeri rappresentanti una caratteristica, da questi dati si costruisce l'insieme di addestramento di cardinalità N:  $D = \{(x_i, y_i)\}_{i=1}^N$ , dove  $y = (y_1, \dots, y_m)$  è l'output dei risultati desiderati e  $y_i$  è l'etichetta. Lo scopo è di apprendere una regola generale che collega i dati in ingresso con quelli in uscita, in modo che l'algoritmo apprenda a classificare un esempio completamente nuovo, non contenente l'etichetta.

Se  $y_i$  è di tipo testuale si parla di classificazione, quando invece è di tipo numerico si parla di regressione. Se indichiamo con C il numero delle classi a cui può appartenere l'output:  $y \in \{1, \dots, C\}$ , se  $C = 2$  la classificazione sarà binaria (in questo caso spesso  $y \in \{0, 1\}$ ); se  $C > 2$  sarà multiclasse.

### 1.2.1 Regressione

La Regressione prevede il valore futuro di un dato, avendo noto il suo valore attuale. Un esempio è la previsione della quotazione delle valute o delle azioni di una società. Nel marketing viene utilizzato per prevedere il tasso di risposta di una campagna sulla base di un dato profilo di clienti; nell'ambito commerciale per stimare come varia il fatturato dell'azienda al mutare della strategia. Questo avviene costruendo una funzione che meglio si adatta ai punti che descrivono la distribuzione delle Y, in funzione delle X, precedentemente osservate. Ovvero: osservati n esempi per cui  $f(x_i) = y_i, \forall i = 1, \dots, n$  si cerca di prevedere il valore di  $\hat{y}$ , dato un nuovo valore  $\hat{x}$ , tramite la stima della funzione  $f$ .

### 1.2.2 Classificazione

La Classificazione viene usata quando è necessario decidere a quale categoria appartiene un determinato dato. Per esempio, data una foto, capire a quale categoria appartiene. In questa tesi vogliamo classificare immagini, più precisamente: capire a quale tipo di monumento corrisponde una determinata immagine.

Questo tipo di algoritmo deve specificare a quale delle k categorie appartiene un input. Viene creata una funzione  $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$  e quando  $y = f(x)$ ,

il modello assegna l'input descritto dal vettore  $x$  ad una categoria identificata dal codice numerico  $y$ .

Esistono altre varianti dell'attività di classificazione, ad esempio, dove  $f$  genera una distribuzione di probabilità su classi.

### 1.3 Apprendimento non supervisionato

Gli algoritmi di apprendimento non supervisionato vengono utilizzati per risolvere problemi di raggruppamento. All'algoritmo viene passato solo l'input:  $D = \{x_i\}_{i=1}^N$  e cerca una relazione tra i dati per capire se e come essi siano collegati tra di loro. Non contenendo alcuna informazione preimpostata, l'algoritmo è chiamato a creare una “nuova conoscenza” (*knowledge discovery*). A differenza del caso supervisionato, questo apprendimento non ha una classificazione o un risultato finale con il quale determinare se il risultato è attendibile, ma generalizza le caratteristiche dei dati e in base ad esse attribuisce ad un input un output: serve generalmente ad estrarre informazioni non ancora note, “creando” esso stesso delle classi in cui dividere i dati, dette *cluster*, da cui prende il nome la tecnica di *clustering*. Si definisce una misura di similarità che se applicata ad un insieme di esempi, descritti da una serie di attributi, partiziona l'insieme in cluster, dove gli esempi appartenenti allo stesso cluster sono simili, mentre esempi appartenenti a cluster differenti sono dissimili. Il problema della tecnica di clustering è trovare la caratteristica per cui si vuole raggruppare l'insieme e trovare una misura a lei adatta.

### 1.4 Apprendimento per rinforzo

Gli algoritmi di apprendimento per rinforzo vengono utilizzati per risolvere problemi di regressione. Lo scopo di questo algoritmo è di realizzare un sistema in grado di apprendere ed adattarsi ai cambiamenti dell'ambiente in cui si trovano, attraverso la distribuzione di una “ricompensa” detta rinforzo, data dalla valutazione delle prestazioni. Questi algoritmi sono costruiti sull'idea che i risultati corretti dovrebbero essere ricordati, per mezzo di un segnale di rinforzo, in modo che diventino più probabili e quindi più facilmente riottenuti nelle volte future; viceversa se il risultato è errato, il segnale sarà una penalità, ovvero si avrà una probabilità più bassa legata a quel determinato output [7].

# Capitolo 2

## Reti neurali artificiali

Le reti neurali sono i modelli di deep learning per eccellenza. Sono un sistema di elaborazione di informazioni ispirato al funzionamento del sistema nervoso umano.

La rete è strutturata come un grafo orientato. I nodi sono raggruppati in strati (*layers*): il primo strato contiene i nodi di input  $x_1, \dots, x_n$ , connessi con lo strato successivo, dove ad ogni arco è associato un peso  $w_i$ . L'ultimo strato contiene i nodi di output. Gli strati tra il primo e l'ultimo strato sono chiamati strati nasconti (*hidden layers*). La lunghezza complessiva del percorso determina la profondità del modello, da cui deriva il nome dell'apprendimento: “deep learning”.

In base all'architettura scelta, esistono vari modelli di reti neurali; la scelta dell'architettura della rete è molto importante, poiché in base al numero di nodi usati per ogni strato ed alla profondità, il costo computazionale cresce o diminuisce: per esempio la scelta di un'architettura poco profonda e con una elevata quantità di nodi per strato, causa un costo computazionale elevato ed un massiccio utilizzo della memoria.

In una rete neurale, ogni neurone artificiale, rappresentato da un nodo, diventa attivo se la quantità totale di segnale che riceve supera la soglia di attivazione, definita dalla cosiddetta funzione di attivazione. Se un nodo diventa attivo, emette un segnale che viene trasmesso lungo i canali di trasmissione fino all'altra unità a cui è collegato.

### Singolo neurone

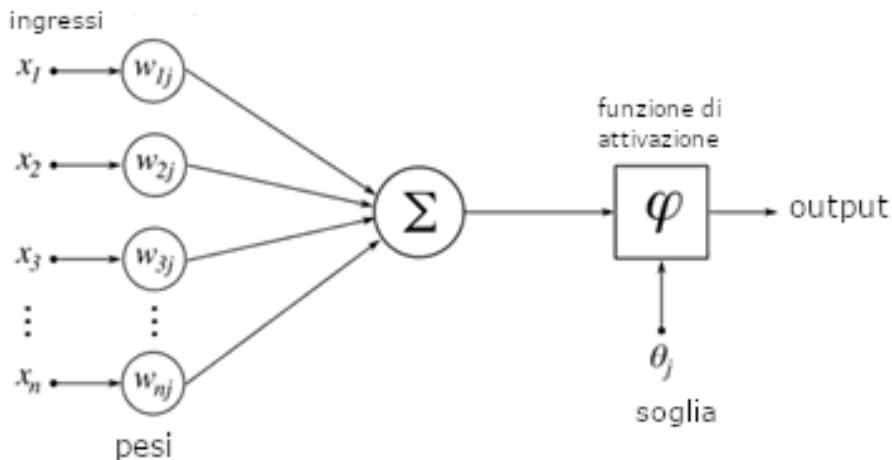


Figura 2.1: Modello non lineare di un neurone artificiale.

La figura 2.1, indipendentemente dal modello di rete utilizzato, mostra l'elaborazione eseguita da un neurone artificiale.

Siano  $x_1, \dots, x_n$  i dati in input, rappresentati dagli  $n$  nodi del primo strato, nel  $k$ -esimo neurone l'informazione viene elaborata come:

$$y_{(k)} = f(b_{(k)} + \sum_{i=1}^n w_{(k)i} \cdot x_{(k-1)i})$$

dove:

- $y_{(k)}$  è l'output generato dal neurone  $k$ ;
- $b_{(k)}$  è il valore soglia del neurone  $k$ ;
- $w_{(k)i}$  è il peso associato all'arco che collega il nodo  $i$ -esimo al neurone  $k$ ;
- $f(\cdot)$  è la funzione di attivazione.

Il motivo principale per cui vengono scelte le reti neurali è la possibilità di parallelizzare i calcoli.

## 2.1 Funzioni di attivazione

La funzione di attivazione è una funzione usata per normalizzare, quindi limitare, l'ampiezza dell' output, per non consumare eccessiva memoria e di velocizzare il processo di calcolo.

Generalmente le reti neurali sono utilizzate per implementare funzioni complesse e le funzioni di attivazione non lineari consentono loro di approssimare funzioni arbitrariamente complesse. Le funzioni maggiormente utilizzate a tale scopo sono 3:

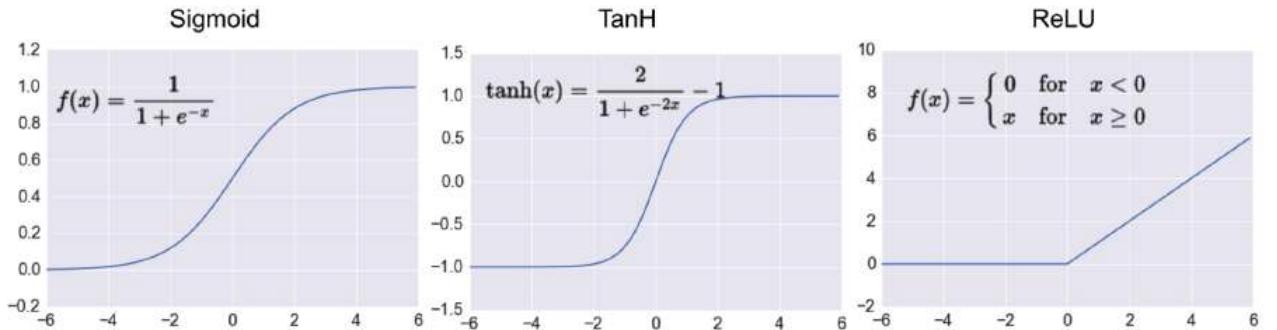


Figura 2.2: Grafici delle funzioni di attivazione più usate.

## 2.2 Addestramento di una rete

Come visto in 1.1, per addestrare un algoritmo si ha bisogno di una funzione di perdita e di un metodo per minimizzare l'errore di valutazione.

L'addestramento di una rete neurale si basa sugli stessi principi:  
si definisce una mappa:

$$y = f(x, \theta)$$

e si cerca il valore del parametro  $\theta$  che più accuratamente approssima la funzione. Bisogna quindi trovare dei parametri che minimizzano la funzione di perdita:

$$L[y, f^*(x, \theta)]$$

ovvero, trovare  $\hat{\theta}$  tali che:

$$\hat{\theta} = \operatorname{argmin}_{\theta} \left\{ \frac{1}{n} \sum_{i=1}^n L[y_i, f^*(x_i, \theta)] \right\}$$

Questo processo è necessario per approssimare una determinata funzione  $f^*$ , che descrive il comportamento della rete.

Durante l'addestramento, la rete viene provata più volte, ogni volta con un input diverso, fino a che l'errore di addestramento è molto piccolo. In questa fase, ad ogni iterazione viene passato in input un insieme di addestramento, viene fissato un valore  $\theta_0$  iniziale, e tramite la funzione di perdita, si calcola l'errore di addestramento. In questo modo la rete ha il valore di quanto ciascun neurone di output sia lontano dal proprio valore atteso, e in che direzione (positiva o negativa).

### 2.2.1 Algoritmo di back propagation

Si consideri una rete neuronale composta da  $L$  strati, dove l'output è composto da un valore solamente. L'algoritmo di back propagation si divide in due fasi:

- propagazione in avanti (*forward propagation*);
- propagazione all'indietro (*backward propagation*)

**Forward propagation** Durante la propagazione in avanti, si calcolano tutti i valori dei nodi presenti nella rete  $a_i^k$ :

$$a_i^k = b_i^k + \sum_{j=1}^{r_{k-1}} w_{ji}^k \cdot o_j^{k-1} \quad (2.1)$$

dove:

- $w_{ij}^k$  è il peso associato all'arco che collega il nodo  $i$  del  $k$ -esimo strato con il nodo  $j$ ;
- $b_i^k$  è il valore soglia del nodo  $i$  nel  $k$ -esimo strato;
- $o_i^k$  è l'output del nodo  $i$  nel  $k$ -esimo strato;
- $r_k$  è il numero di nodi presenti nel  $k$ -esimo strato.

Calcolato l'ultimo valore  $a^L$ , corrispondente all'output della rete, si esamina l'errore di allenamento tramite la funzione di perdita. Per minimizzare l'errore, basta minimizzare la funzione di perdita, questa operazione sancisce l'inizio della seconda fase: la backward propagation.

**backward propagation** Per ottenere il minimo di una funzione bisogna

vedere dove si annulla la sua derivata prima. Trattandosi di una funzione a più variabili, si dovrà calcolare il gradiente della funzione di perdita:  $L[y, f(x, \theta)]$ , per farlo si applica la regola della catena per il calcolo e poiché  $\theta = (W^1, W^2, \dots, W^{L-1})$ , abbiamo:

$$\frac{\partial L}{\partial w_{ij}^k} = \frac{\partial L}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k}. \quad (2.2)$$

Il punto di forza di questo algoritmo consiste nell'introdurre una quantità di costo (anche chiamata errore):

$$\delta_j^k \equiv \frac{\partial L}{\partial a_j^k} \quad (2.3)$$

attraverso la quale si calcolano le derivate parziali in modo iterativo, ripercorrendo la rete all'indietro.

Denotando con  $g$  la funzione di perdita dello strato nascosto, si ottiene la formula di propagazione all'indietro:

$$\delta_j^k = g'(a_j^k) \sum_{l=1}^{r_{k+1}} w_{jl}^{k+1} \delta_l^{k+1} \quad (2.4)$$

e la derivata parziale della funzione di perdita, rispetto ai pesi degli strati nascosti  $w_{ij}^{k+1}$ ,  $1 \leq k < L$ , si ottiene con:

$$\frac{\partial L}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'(a_j^k) o_i^{k-1} \sum_{l=1}^{r_{k+1}} w_{jl}^{k+1} \delta_l^{k+1} \quad (2.5)$$

Ora può avvenire l'aggiornamento dei pesi tramite il metodo SGD (Stochastic Gradient Descent), ovvero in formule:

$$w_{ij}^{k+1} = w_{ij}^k + \eta \frac{\partial L}{\partial w_{ij}^k} \quad (2.6)$$

dove  $\eta \in (0, 1]$  rappresenta il fattore di apprendimento (*learning rate*).

La scelta del fattore di apprendimento influenza molto il comportamento dell'algoritmo in quanto, se si scelgono valori troppo piccoli la convergenza sarà lenta, mentre se si scelgono valori troppo grandi si rischia di avere una rete instabile con comportamento oscillatorio.

Un metodo semplice per incrementare il fattore di apprendimento, senza il rischio di rendere la rete instabile, è quello di modificare la regola di aggiornamento inserendo nell'equazione 2.6 un ulteriore parametro  $\alpha$  (detto momento), ottenendo:

$$w_{ij}^{k+1} = \alpha w_{ij}^k + \eta \frac{\partial C}{\partial w_{ij}^k}. \quad (2.7)$$

Se si espande ricorsivamente la formula si ottiene:

$$w_{ij}^{k+1} = \eta \sum_{l=1} k + 1 \alpha_{k+1-l} \frac{\partial L}{\partial w_{ij}^l} \quad (2.8)$$

Inserendo il momento si ha il vantaggio che se la derivata parziale tende a mantenere lo stesso segno su iterazioni consecutive, grazie alla sommatoria, l'aggiornamento produce valori più ampi e quindi tende ad accelerare nelle discese del gradiente. Se invece la derivata ha segni opposti ad iterazioni consecutive, la sommatoria tende a diminuire l'ampiezza dell'aggiornamento, in questo modo non si corre il rischio di avere dei loop infiniti nel caso di minimi locali della funzione d'errore.

A tal proposito generalmente i criteri di arresto dell'algoritmo possono essere:

- $\|\nabla L\| < \epsilon$
- $L[y, f(x, \theta)] = 0;$
- $L_i[y, f(x, \theta_i)] - L_j[y, f(x, \theta_j)] << \epsilon'$

dove  $L_i$  e  $L_j$  sono due epocha consecutive. Un'epoca corrisponde alle due fasi di propagazione necessarie per un aggiornamento dei pesi.

## Il problema del Vanish Gradient

Come accennato in precedenza, l'uso del metodo del gradiente va applicato con attenzione e parametri adatti, altrimenti si incorre nel fenomeno detto “sparizione del gradiente”, ovvero del *vanish gradient*.

Si ipotizzi di avere una rete profonda semplice, ovvero con 3 livelli nascosti e con un solo neurone in ogni strato:

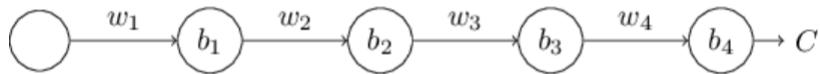


Figura 2.3: Rete neurale semplice costituita da 3 strati con:  $w_j$  pesi,  $b_j$  i valori soglia e  $C$  funzione di costo.

Applicando l'eq 2.1 per il calcolo dell'output di un nodo, alla rete rappresentata in figura 2.3, si ottiene:

$$a_j = \sigma(w_j a_{j-1} + b_j) \quad (2.9)$$

dove  $\sigma$  è la funzione di attivazione Sigmoid.

Si è indicato con  $C$  una funzione di costo, per sottolineare che il costo è una funzione dell'output della rete (in questo caso  $a_4$ ).

Se  $C$  ha un valore basso l'output sarà vicino a quello desiderato, viceversa per alti valori di  $C$  l'output calcolato si allontanerà dal risultato che ci si auspicava.

Il gradiente associato al primo strato viene calcolato applicando la regola della catena ottenendo la seguente formula:

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} w_4 \sigma'(z_4) w_3 \sigma'(z_3) w_2 \sigma'(z_2) \sigma'(z_1). \quad (2.10)$$

Si noti che l'equazione 2.10 è il prodotto di termini della forma:  $w_j \sigma'(z_j)$ , fatta eccezione per il primo termine.

Tipicamente, i valori iniziali dei pesi vengono ricavati dai valori che può assumere una distribuzione normale avente media 0 e deviazione standard 1, allora:

$$|w_j| < 1, \quad \forall j.$$

La derivata raggiunge il massimo in 0:  $\sigma'(0) = \frac{1}{4}$ , quindi ogni termine dell'equazione è tale che

$$|w_j \sigma'(z_j)| < \frac{1}{4}, \quad \forall j,$$

si ottiene la stima del tipo:

$$\frac{\partial C}{\partial b_1} = \frac{\partial C}{\partial a_4} \underbrace{w_4 \sigma'(z_4)}_{< \frac{1}{4}} \underbrace{w_3 \sigma'(z_3)}_{< \frac{1}{4}} \underbrace{w_2 \sigma'(z_2)}_{< \frac{1}{4}} \underbrace{\sigma'(z_1)}_{< 1}$$

Questo comportamento implica che, con l'aumentare della profondità della rete, il valore del gradiente tende a diminuire di un fattore di  $\frac{1}{4}$  per ogni strato, fino a tendere a 0 prematuramente; ovvero a "scomparire" da un certo strato in poi.

L'utilizzo della funzione di attivazione ReLu risolve il problema, grazie alla sua caratteristica di avere il gradiente sempre pari a 1. Questo comporta che, non solo il gradiente non subisce una diminuzione progressiva in ogni strato, ma mantiene il suo valore inalterato.

$ReLu(z) = z_+ = \max(0, z)$ , il che comporta che alcuni neuroni non vengono attivati e quindi non influenzano in alcun modo l'algoritmo scelto per individuare i pesi ottimali per la rete.

Ne consegue che  $ReLu'(z) = 1$  per ogni nodo rimasto attivo.

## 2.3 Reti Deep Feed-forward

Le reti Feed-forward sono le reti neurali profonde con la struttura più semplice, composte da almeno uno strato nascosto. La loro struttura è rappresentata da un grafo aciclico diretto in un'unica direzione, dove ogni nodo di uno strato è connesso con tutti i nodi dello strato successivo e nessun nodo è connesso con un nodo appartanente allo stesso strato.

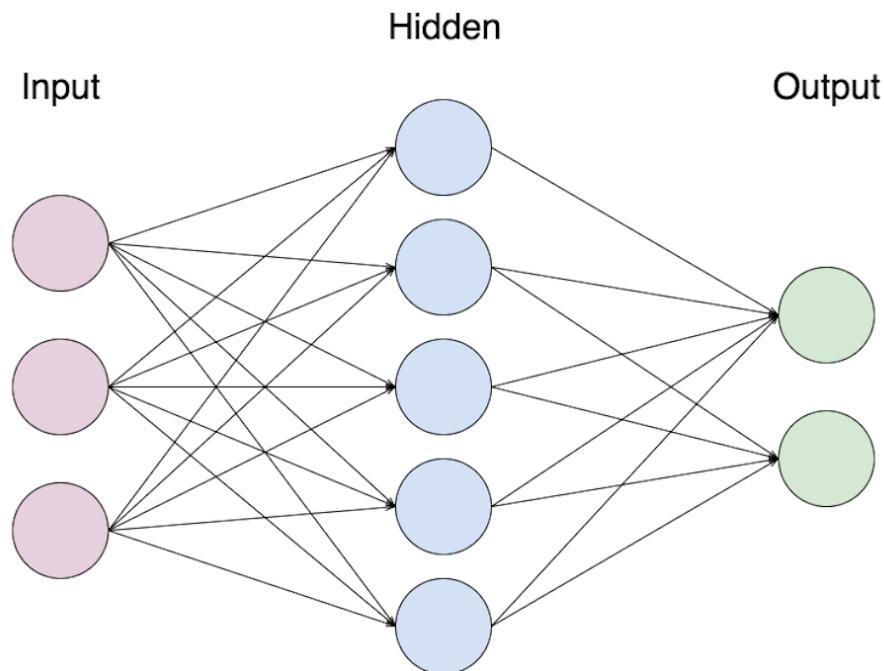


Figura 2.4: modello di rete deep feedforward con uno strato

Questo tipo di reti sono di solito rappresentate dalla composizione di più funzioni, dal momento che l'output di un nodo non può essere passato in input ad una funzione già eseguita. Nel caso di una rete con  $L$  strati avremo che la relazione che lega il vettore di input  $x$  con quello di output  $y$  risulta essere:

$$y = f(x, \theta) = g_o(\theta_{L-1}g_{L-1}(\dots\theta_2g_2(\theta_1x))) \quad (2.11)$$

Lo scopo della rete è sempre di approssimare  $f(x)$  ad una funzione  $f^*$  e questo è possibile grazie all'addestramento.

## 2.4 ImageNet

In Internet oramai sono presenti milioni di milioni di immagini e di video, usati per i modelli e gli algoritmi più sofisticati e robusti, per aiutare gli utenti ad indicizzare, recuperare, organizzare e interagire con questi dati. Un esempio banale è la ricerca di immagini su Google.

Di solito i motori di ricerca possono trovare una determinata immagine solo se il testo inserito corrisponde al testo con cui è stato etichettato.

È stato creato *ImageNet*: un grande database visivo contenente oltre 14 milioni di immagini etichettate, progettato per l'utilizzo nella ricerca di software per il riconoscimento di oggetti visivi.

*ImageNet* si basa sulla struttura gerarchica fornita da un altro database: *WordNet* [8].

Un'immagine digitale è formata da diversi quadratini disposti in modo regolare su una griglia di punti equidistanti. Questi quadratini sono detti *pixel* (*picture elements*) e determinano la dimensione e la risoluzione di un'immagine. In ogni pixel risiede un'informazione espressa in bit riguardante (generalmente) il colore, che rappresenta il livello di intensità dei colori fondamentali.

Il modello più utilizzato è quello RGB, dove per ogni pixel vengono utilizzati 3 byte:

- 1 byte per la componente rossa (R)
- 1 byte per la componente verde (G)
- 1 byte per la componente blu (B)

ma ne esistono anche altri come, ad esempio, CMYK che considera come colori fondamentali: ciano, magenta, giallo e nero.

Tipicamente nel dataset di ImageNet la dimensione media delle immagini è di circa  $400 \times 350$  pixel.

## 2.5 Reti Neurali Convoluzionali - CNNs

Le reti neurali convoluzionali (*Convolutional Neural Networks*) sono un tipo particolare di reti neurali a strati che usano, in almeno uno dei loro strati, la convoluzione al posto della classica moltiplicazione tra matrici.

**Definizione 6** Siano  $f, g \in L^1(\mathbb{R})$ , si definisce convoluzione tra  $f$  e  $g$  come:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau = y(t) \quad (2.12)$$

Tramite l'eq. 2.12, se si conosce la funzione di risposta  $g(t)$  e il segnale di entrata  $f(t)$ , si può esprimere il segnale di uscita  $y(t)$ , che nelle reti convoluzionali rappresenta il filtro di convoluzione (*feature map*) e la funzione di risposta è detta “*kernel*”. Si ricorda che per *segnale* si intende una grandezza fisica qualsiasi a cui è associata un'informazione.

Nelle reti neurali i segnali non sono descritti da funzioni continue ma da funzioni discrete, in quanto l'input passato ad un neurone è un valore discreto ben preciso.

Si necessita quindi delle seguenti definizioni:

**Definizione 7** *I segnali discreti sono definiti come funzioni di variabili indipendenti che possono assumere solo un insieme finito di valori discreti. Un segnale discreto nel tempo  $x$ , consiste in una sequenza di numeri indicata con*

$$x_n \quad \text{oppure} \quad x(n), \quad n \in \mathbb{Z}$$

**Definizione 8** *Siano  $f(n)$  e  $g(n)$  due sequenze, si definisce la loro convoluzione discreta come:*

$$f(n) * g(n) = \sum_{k=-\infty}^{\infty} x(k)g(n-k) = y(k) \quad (2.13)$$

Nel nostro problema di analisi di immagini, data un'immagine di dimensione  $M \times N$  pixel l'eq2.13 diventa:

$$(f * g)(x, y) = \sum_{i=0}^{M} \sum_{j=0}^{N} f(x, y)g(x-i, y-j) \quad (2.14)$$

La caratteristica principale per cui le reti convoluzionali sono maggiormente indicate per l'elaborazione di immagini è la loro struttura, basata sull'elaborazione di dati nella forma di array multipli.

Come descritto nel paragrafo 2.4 è facilmente intuibile come ogni dimensione caratterizzante un'immagine può essere vista come un array multidimensionale, ovvero un tensore.

I neuroni di ciascun livello sono organizzate in griglie o volumi 3D.

L'altezza e la larghezza rappresentano i pixel, mentre la profondità le caratteristiche dell'oggetto, chiamate *feature map*.

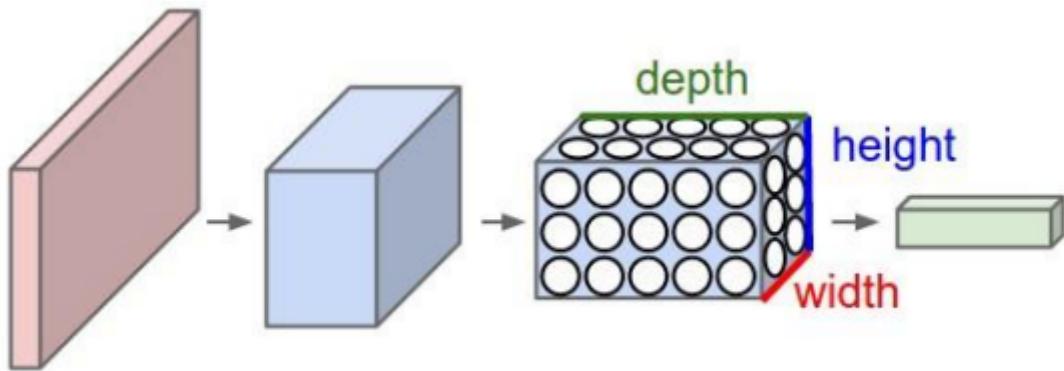


Figura 2.5: Esempio di rappresentazione di un’immagine in formato RGB in una CNN.

### 2.5.1 Struttura di una CNNs

Alla base di questa tipologia di reti neurali ci sono 3 concetti:

1. connettività locale (*sparse interactions*)
2. condivisione dei parametri
3. alternanza di strati di convoluzione e di *pooling*

La **connettività locale** consiste nello sfruttare le correlazioni spaziali presenti all’interno dei dati di input e viene applicata tra neuroni di strati adiacenti. Questo è possibile applicando un kernel più piccolo dell’input che, tramite l’operazione di convoluzione, darà in output un numero minore di neuroni. Ne consegue una forte riduzione del numero di connessioni tra i nodi della rete.

Nelle prossime sezioni sarà più chiaro cosa questo voglia dire, ma se si considera, ad esempio, un’immagine e su di essa ci si focalizzi su un singolo pixel, in base alle correlazioni con i pixel circostanti si possono ricavare informazioni locali sull’immagine (ad esempio bordi, colori, forme geometriche, ecc...). Attraverso la **condivisione dei parametri** si utilizza lo stesso parametro per più di una funzione in un modello, permettendo alla rete di apprendere un insieme di parametri invece che insiemi separati di parametri per ogni posizione, questo comporta che l’algoritmo deve tenere in memoria un numero minore di parametri.

L'architettura di una rete convoluzionale è formata, tipicamente, da tre tipi di strati:

1. convoluzionali;
2. pooling;
3. totalmente connessi.

Uno strato tipico di una rete convoluzionale consiste di tre fasi .Nella prima fase lo strato esegue diverse convoluzioni in parallelo per produrre un set di attivazioni lineari. Nella seconda fase, ogni attivazione lineare viene eseguita attraverso una funzione di attivazione non lineare. Nella terza fase, si esegue una funzione di pooling per modificare ulteriormente l'output dello strato.

### Strati convoluzionali

Lo scopo dello strato convoluzionale (*convolutional layer*) è quello di rilevare delle particolari caratteristiche all'interno di un oggetto, grazie ad un'analisi locale. Questo compito è affidato al kernel della convoluzione, detto anche filtro, che si comporta come una finestra bidimensionale composta da pesi che scorre su tutto l'input. Di ogni porzione di input viene calcolata la sua convoluzione con il kernel, producendo così un output di dimensioni ridotte rispetto all'input. Questo output viene chiamato *feature map* e racchiude le caratteristiche che il filtro cercava ed è qui che i pesi sono condivisi: i neuroni di una stessa feature map processano porzioni diverse dell' input nello stesso modo.

Il kernel ha due iperparametri, chiamati passo (*stride*) e dimensione (*size*). La size può essere qualsiasi dimensione di un rettangolo, mentre lo stride è il numero di pixel che si fanno scorrere tra due computazioni della finestra di kernel. Se lo stride ha lunghezza 1, il filtro si sposta di un pixel alla volta, se ha lunghezza 2, i filtri saltano 2 pixel alla volta durante lo scorrimento, producendo un'immagine con dimensione dimezzata.

Un altro modo per regolare la dimensione delle feature map è tramite l'aggiunta di un bordo al volume di input, inserendo tutti valori nulli in prossimità del bordo. Con il parametro *Padding* si denota lo spessore in pixel del bordo. L'uso di questo parametro ha il vantaggio di mantenere le dimensioni spaziali costanti dopo lo strato convoluzionale, migliorando le prestazioni. Questo perché se gli strati non azzerassero gli input e realizzassero solo convoluzioni valide, allora la dimensione dei volumi si ridurrebbe di una piccola quantità ad ogni convoluzione, in questo modo le informazioni ai bordi si perderebbero troppo rapidamente.

Sia  $W_{out}$  la dimensione orizzontale (verticale) delle feature map di output e  $W_{in}$  la corrispondente dimensione nell'input. Sia inoltre  $F$  la dimensione (orizzontale) del filtro e  $K$  il numero di filtri da utilizzare.

Vale la seguente relazione:

$$W_{out} = \frac{W_{in} - F + 2 \cdot Padding}{Stride} + 1 \quad (2.15)$$

e la profondità dell'output sarà pari a  $K$ .

### Strati di pooling

Lo scopo degli strati di pooling è quello di unire caratteristiche simili (in quanto vicine) in una sola, per ridurre progressivamente la dimensione spaziale della rappresentazione. Di conseguenza si riduce anche la quantità di parametri, che potebbero comportare un sovraccarico della rete.

Ogni strato appartenente alla profondità dell'input viene suddiviso in piccoli quadrati, chiamati *subsampling*, di dimensione  $r \times r$ , con  $r \in \mathbb{N}$  e per ogni quadrato viene calcolato indipendentemente o il valore massimo (Max) o la media (Avg) dei suoi neuroni.

Max e Avg sono gli operatori di aggregazione maggiormente utilizzati e sono invarianti per piccole traslazioni. L'invarianza per traslazione è molto utile in quanto se una caratteristica è interessante in una parte dell'immagine, presumibilmente lo sarà anche se è posizionata in un altro spazio.

Passata in input un'immagine di volume  $W_{in} \times H_{in} \times D$ , lo strato di pooling di dimensione  $F \times F$  e stride  $S$  produce un volume di dimensioni  $W_{out} \times H_{out} \times D$ , dove:

$$W_{out} = \frac{W_{in} - F}{S} + 1 \quad (2.16)$$

Alternando ripetutamente queste due tipologie di strati, si arriva ad una configurazione spaziale unita dell'immagine di piccole dimensioni. In quest'ultima prospettiva si passa a strati totalmente connessi, specialmente nell'ultimo strato dove l'output deve necessariamente essere una delle possibili classi.

# Capitolo 3

## Shazarch

L'obiettivo di questa tesi, come detto più volte, è di trovare l'architettura ottimale per un modello di Deep Learning capace di riconoscere, quindi classificare, immagini di resti e monumenti presenti nell'area archeologica del Foro Romano a Roma. Vista la difficoltà, anche per un occhio umano, di riconoscere alcuni reperti si è pensato di inserire anche le informazioni relative alla geolocalizzazione di questi ultimi.

Questo modello è finalizzato alla creazione di un'applicazione Android, chiamata Shazarch, in grado di permettere al turista in visita al Foro Romano, di inquadrare l'oggetto di interesse e sapere cosa sta visualizzando in quel momento. Per fare questo si necessita di un modello di classificazione di deep learning, addestrato con un dataset di immagini provenienti dal sito archeologico e delle coordinate geografiche dei singoli monumenti. Si sta parlando di un algoritmo di apprendimento supervisionato, quindi il dataset che si utilizza deve contenere delle variabili rappresentanti le caratteristiche dell'oggetto da riconoscere, tra cui naturalmente anche un'etichetta. L'etichetta corrisponde alla classe di appartenenza dell'oggetto che, nel nostro caso, corrisponde ad un numero  $n \in \{0, \dots, 29\}$ , in quanto i reperti che la rete è in grado di riconoscere sono 30.

Per quanto riguarda le altre variabili si sono considerati i valori dei pixel delle immagini in formato RGB e di dimensione  $3024 \times 4032$ .

### Costruzione dataset

Il dataset di immagini deve essere abbastanza grande per far sì che l'algoritmo abbia abbastanza immagini su cui addestrarsi e anche per verificare la sua accuratezza, si ricorda infatti che l'insieme di addestramento e di test(e/o di validazione) formano una partizione dell'insieme totale di partenza.

Il dataset utilizzato è composto da circa 40000 immagini, di cui una piccola parte originali, ovvero scattate fisicamente sul posto, le restanti sono state generate artificialmente tramite la libreria Augmentor. Successivamente l'in-

sieme di dati è stato diviso in modo che l'80% andasse a costituire l'insieme di addestramento e il 20% quello di validazione. Come architettura per il modello si è optato per l'architettura pre-esistente mobileNet, descritta nel dettaglio nella sezione 3.1, addestrata tramite il metodo di discesa del gradiente (SGD) settato con il parametro di learning rate=0.1, dove nella fase di back propagation (2.2.1) batch size=100. Una volta addestrata la rete, tensorflow salva il modello all'interno di un file, successivamente convertito con estensione “.lite”. Questa conversione trasforma il modello in uno di tipo TensorFlow Lite, un formato che permette di utilizzare Android Studio: un ambiente di sviluppo integrato (IDE) per lo sviluppo per la piattaforma Android. All'interno del codice Java, linguaggio con cui si programma un'applicazione Android, sono state inserite le coordinate di geolocalizzazione.

Sono stati implementati due metodi per considerare le posizioni GPS del visitatore. Ad ogni metodo corrisponde un'applicazione. Una, chiamata “GPS”, considera la distanza tra il dispositivo ed il monumento osservato.

L'altra applicazione sviluppata, chiamata “GPS Alarm”, invece considera una circonferenza di raggio 20metri, centrata in un monumento che ne delimita la sua area “protetta”. Se il visitatore non si trova all'interno di quest'area la probabilità di osservare il monumento corrispondente ad essa è pari a zero, altrimenti è 1. Questo solo per poter considerare solo i reperti che è ragionevole pensare un turista possa voler inquadrare. Naturalmente questi approcci non permettono di classificare un monumento da una distanza maggiore di quella considerata, anzi produrrà una classificazione errata, in quanto verranno considerate solo i monumenti in cui le coordinate del visitatore si trovano all'interno delle loro aree protette.

Ora che sono note la struttura e la logica con cui è stata progettata Shazarch, si possono approfondire alcuni aspetti accennati in questa fase.

### 3.1 MobileNet

MobileNet è un'architettura rilasciata da Google e resa disponibile open source, contenente una famiglia di modelli di computer vision per dispositivi mobili per TensorFlow. La particolarità di questa architettura è l'uso di convoluzioni separabili per costruire reti neurali profonde leggere.

Le convoluzioni separabili sono delle convoluzioni formate da due convoluzioni: una convoluzione in profondità è una convoluzione con un filtro di dimensione  $1 \times 1$  chiamata convoluzione puntuale. La convoluzione in profondità applica un singolo filtro a ciascun canale di ingresso e all'output generato viene applicato il filtro della convoluzione puntuale. Riassumendo: la convoluzione standard consente di filtrare e combinare gli input in un nu-

vo set di output in un unico passaggio, mentre la convoluzione separabile lo fa in due passaggi, ognuno svolto da uno strato di convoluzione diverso.

Questa fattorizzazione ha l'effetto di ridurre drasticamente il calcolo e la dimensione del modello. Uno strato convoluzionale standard ha un costo computazionale pari a:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

mentre nella convoluzione separabile:

$$D_K \cdot D_k \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

Se si fa il rapporto dei costi computazionali dei due approcci, si ottiene una riduzione del calcolo di:

$$\frac{D_k \cdot D_k \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_k \cdot D_k \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2} \quad (3.1)$$

L'architettura di MobileNet è quindi costituita da un primo strato con una convoluzione standard e da 12 convoluzioni separabili tutti implementati con una funzione di attivazione ReLu con batchnorm.

Dopo questi strati è presente un altro strato di convoluzione separabile, seguito da uno strato di Avg pooling (calcola la media dei suoi neuroni) che riduce la risoluzione spaziale ad 1 ed infine lo strato di output fortemente connesso con funzione di attivazione Softmax per la classificazione. Ricapitolando, l'architettura ha:

- primo strato con convoluzione standard;
- 26 strati nascosti con convoluzione separabile;
- ultimo strato di output forteente connesso.

per un totale di 28 strati, in quanto ogni strato nascosto è composto da una convoluzione separabile, che corrisponde a due strati consecutivi (convoluzione di profondità e convoluzione puntuale).

A partire da questa architettura si possono costruire modelli più piccoli e meno dispendiosi dal punto di vista computazionale, introducendo un “moltiplicatore di larghezza”  $\alpha$  e un “moltiplicatore di risoluzione”  $\rho$ .

Il ruolo del moltiplicatore di larghezza è quello di assottigliare uniformemente una rete su ciascun livello, mentre il moltiplicatore di risoluzione  $\rho$  serve a ridurre la dimensione dell'immagine. Viene infatti applicato all'immagine di

input in modo che la rappresentazione interna di ogni livello viene successivamente ridotta dallo stesso moltiplicatore. Aggiungendo questi due parametri il nuovo costo computazionale della rete è:

$$D_K \cdot D_K \cdot \alpha M \cdot \rho D_F \cdot \rho D_F + \alpha M \cdot \alpha N \cdot \rho D_F \cdot \rho D_F \quad (3.2)$$

con  $\alpha, \rho \in (0, 1]$  [9].

Per creare Shazarch è stata utilizzata l'architettura “standard” di mobileNet, ovvero dove gli iperparametri  $\alpha$  e  $\rho$  sono uguali a 1.

Inoltre si ricorda che la dimensione delle foto, con cui era stato costruito il dataset, era di  $3024 \times 4032$  pixels, mentre imageNet accetta solo immagini di dimensioni  $224 \times 224$ ,  $192 \times 192$ ,  $160 \times 160$  o  $128 \times 128$  pixels, quindi le immagini del dataset sono state ridimensionate in modo da soddisfare la dimensione richiesta. Questo ha comportato una perdita parziale di informazioni sulle immagini, ma era un prezzo da pagare per creare un'applicazione adattabile su un dispositivo mobile che non dispone di una memoria molto capiente.

## 3.2 TensorFlow

TensorFlow (TF) è una libreria software open source, sviluppata da Google, utilizzata per implementare l'apprendimento automatico e i sistemi di deep learning. Essa fornisce API native in linguaggio: Python, C/C++, Java, Go, e RUST.

Per questo lavoro è stato utilizzato il linguaggio di programmazione Python. In generale un algoritmo scritto in TF rispetta la seguente struttura:

1. Importare ed analizzare l'insieme di dati;
2. Creare colonne di caratteristiche per descrivere i dati;
3. Selezionare il tipo di modello;
4. Provare il modello;
5. Valutare l'efficacia del modello;
6. Lasciare che il modello addestrato faccia previsioni (test).

Questa struttura è in linea con la descrizione di un generico algoritmo di apprendimento, descritto in (1.1). La vera innovazione di TF risiede nella descrizione del modello, poiché lo fa costruendo un grafico computazionale: *Data Flow Graph*. In questo grafico ogni nodo rappresenta l'istanza di un'operazione matematica, mentre ogni spigolo rappresenta un tensore, su cui vengono eseguite le operazioni.

**Definizione 9** Un **tensore** in TF è una matrice  $n$ -dimensionale di tipi di dati di base (es: *float32*, *int32*, *string*, ecc..). Viene chiamato *tf.Tensor* ed è descritto da tre parametri:

1. *grado (rank)*;
2. *corpo (shape)*;
3. *tipo (type)*.

Il **grado** di un oggetto *tf.Tensor* è il suo numero di dimensioni.

Il **corpo** di un *tf.Tensore* è il numero di elementi in ogni dimensione. TF automaticamente deduce il corpo durante la costruzione del grafico.

Il **tipo** è il tipo di dato a cui appartengono gli elementi del tensore.

I principali tipi di tensori sono:

- Variabili (*tf.Variable*): i parametri dell'algoritmo che verranno cambiati per ottimizzare l'algoritmo;
- Costanti (*tf.constant*);
- Segnaposto(*tf.placeholder*): consentono di inserire dati e di creare operazioni per costruire il grafico computazionale, possono dipendere da altri dati ad esempio il risultato previsto di un calcolo. Possono essere usati più volte e non dare lo stesso risultato;
- Tensore sparso (*tf.SparseTensor*).

### 3.2.1 Shazarch

Di seguito si riporta il codice utilizzato per creare il modello per l'applicazione Shazarch.

Per prima cosa si è ampliato il dataset di partenza, tramite la generazione di foto artificiali con Augmentor [10].

Una volta creato il dataset, si crea l'elenco delle etichette, che in questo caso corrispondono al nome delle cartelle che contenevano le immagini. Si crea un dizionario che associa un numero ad una classe. Si ridimensionano le immagini e si crea il modello mobileNet:

```
base_model=MobileNet(input_shape=(224,224,3), include_top=False,
                      input_tensor=Input(shape=(224,224,3)), pooling=None)
```

dove:

- *input\_shape* è il formato delle immagini passate in input alla rete;

- include\_top specifica se includere lo strato completamente connesso nella parte iniziale della rete;
- input\_tensor è la dimensione dell'input dell'immagine del modello;
- pooling specifica se applicare il pooling finale, si può settare solo se include\_top=False.

Per questa configurazione sono stati settati  $\alpha, \rho = 1$ , che rappresentano rispettivamente il moltiplicatore di larghezza e di risoluzione. Non compaiono tra gli argomenti perchè, se omessi, sono settati già ad 1 di default. Discorso analogo per il valore relativo al dropout, che di default vale 0.001.

Si definiscono le ultime operazioni da applicare all'output della rete, questo passaggio può essere inteso come un ultimo strato che, va a completare l'architettura. Infine si crea il modello e lo si allena.

Ricapitolando, di seguito viene riportato l'intero codice:

```
from __future__ import print_function
import os
import matplotlib.image as mpimg
from os import listdir
from os.path import join
import numpy as np
import glob
import tensorflow as tf
from keras.models import Model
from keras.applications.mobilenet import preprocess_input, decode_predictions
from keras.applications.mobilenet import MobileNet
from keras.layers import Dense, GlobalAvgPool2D, Input, Dropout
from keras.callbacks import ModelCheckpoint, CSVLogger
from keras.callbacks import ReduceLROnPlateau, LearningRateScheduler
from keras.optimizers import SGD
from keras.regularizers import l2
import keras.backend as K
from keras.utils.np_utils import to_categorical

_files = glob.glob('/percosdo/della/cartella/dataset/*')
labels = []
for f in _files:
    labels.append(f.split('/')[-1])
print(labels)
with open('/percordo/dove/salvare/file/label.txt','a+') as f:
    for each in labels:
        f.write(each+'\n')
f.close()
assert(len(labels)==30)

train_path = '/percosdo/della/cartella/Train/'
```

```

valid_path = '/percosdo/della/cartella/Validation/'

class_to_ix = {}
ix_to_class = {}
classes = [l.strip() for l in labels]
class_to_ix = dict(zip(classes, range(len(classes))))
ix_to_class = dict(range(len(classes)), classes)
class_to_ix = {v:k for k,v in ix_to_class.items()}

def load_images(path):
    all_imgs = []
    all_classes = []
    for i, subdir in enumerate(listdir(path)):
        imgs = listdir(join(path,subdir))
        class_ix = class_to_ix[subdir]
        print("{} - {} - {}".format(i, class_ix, subdir))
        for img_name in imgs:
            img_rr = mpimg.imread(join(path,subdir,img_name))
            if img_rr.shape==(224,224,3):
                all_imgs.append(img_rr)
                all_classes.append(class_ix)
            else:
                print("ERROR")
    print("All images : {}".format(len(all_imgs)))
    return np.array(all_imgs),np.array(all_classes)

X_train, Y_train = load_images(train_path)
X_val, Y_val = load_images(valid_path)

n_classes = len(labels)
n_classes
Y_train_cat = to_categorical(Y_train, n_classes)
Y_val_cat = to_categorical(Y_val, n_classes)

session = tf.Session()
K.set_session(session)
base_model=MobileNet(input_shape=(224,224,3),include_top=False,
                      input_tensor=Input(shape=(224,224,3)),pooling=None)
out = base_model.output
out = GlobalAvgPool2D(name='avg-pools26')(out)
out = Dropout(0.4)(out)
y_pred=Dense(n_classes,activation='softmax',kernel_initializer='random_uniform',
              kernel_regularizer=l2(.0005),name='last_layers26')(out)
model = Model(inputs=base_model.input,outputs=y_pred)
optimizer = SGD(lr=.01,momentum=.9)
model.compile(optimizer=optimizer,loss='categorical_crossentropy',
              metrics=['accuracy'])
checkpointer=ModelCheckpoint(filepath='/percosdo/dove/salvare/file/model.{epoch:02d}-{val_loss:.2f}.hdf5', verbose=1, save_best_only=True)

```

```

csv_logger = CSVLogger( '/percosdo/dove/salvare/file/model.log' )
def schedule(epoch):
    if epoch < 15:
        return .01
    elif epoch < 28:
        return .002
    else:
        return .0004

lr_scheduler = LearningRateScheduler(schedule)
model.fit(X_train, Y_train_cat, batch_size=100, epochs=10000, verbose=2,
           callbacks=[lr_scheduler, csv_logger, checkpointer],
           validation_data=(X_val, Y_val_cat), shuffle=True)

```

Il modello appena creato è di tipo tensorflow, ma per poterlo esportare, in modo che possa essere sfruttato, ha bisogno di essere convertito. In questo caso si necessita di un formato di tipo tensorflowLite, che è compatibile con AndroidStudio: l'ambiente di sviluppo scelto per creare l'applicazione. Nel prossimo paragrafo verrà illustrato l'utilizzo di AndroidStudio.

### 3.3 AndroidStudio

Android Studio è un ambiente di sviluppo integrato (IDE), open source, per lo sviluppo per la piattaforma Android.

Alla base di ogni applicazione Android ci sono quattro tipi di componenti: Activity, Service, Content Provider e BroadcastReceiver.

Un **Activity** è un'interfaccia utente. Ogni volta che si usa un'App, generalmente, si interagisce con una o più schermate, mediante le quali si consultano dati o si immettono input. Essa è il punto di partenza di ogni applicazione ed è la componente con cui l'utente ha il contatto più diretto.

Un **Service** svolge un lavoro, generalmente lungo e continuato, che viene svolto interamente in *background* senza bisogno di interazione diretta con l'utente. I Service hanno un'importanza basilare nella programmazione proprio perché, spesso, preparano i dati che le activity devono mostrare all'utente, permettendo una reattività maggiore nel momento della visualizzazione.

Un **Content Provider** nasce con lo scopo della condivisione di dati tra applicazioni, permettendo di condividere, nel sistema, contenuti custoditi in un database, su un file o reperibili mediante accessi in Rete.

Un **Broadcast Receiver** è un componente che reagisce ad un invio di messaggi a livello di sistema, con cui Android notifica l'avvenimento di un determinato evento, ad esempio l'arrivo di un SMS o di una chiamata. Questi componenti sono utili per la gestione istantanea di determinate circostanze speciali.

Un **Intent** è un oggetto che associa due componenti separate, come due attività, in seguito dell’invocazione di una per l’altra. Esso rappresenta infatti l’intento di fare qualcosa, senza che la componente chiamante cessi la sua esistenza. Un **Fragment** è strettamente legato alla propria activity, dalla quale riceve eventi in input, che possono essere elaborati mentre l’attività è in esecuzione. La particolarità di questo componente risiede nel fatto che, un Fragment può avere un’interfaccia utente totalmente sua, sempre però all’interno di quella dell’activity. Si può pensare ad esso come un frammento dell’Activity, indipendente da essa, un esempio è dato da i menu a tendina al lato della schermata di un’applicazione.

All’interno di AndroidStudio il linguaggio utilizzato è quello Java, per le parti delle componenti principali. Mentre le parti che descrivono l’ “estetica” dell’applicazione, ovvero i *layout*, sono scritti in .xml.

Quando si inizia un progetto Android, la prima cosa da fare è istanziare l’activity principale, senza la quale non può essere costruita un’applicazione.

La mainActivty, dal punto di vista del codice, è una classe che estende la super classe Activity e dalla quale eredita dei metodi che verranno implementati tramite *override*. In generale ciò che collegherà i layer con la classe sarà il documento, in formato .xml, chiamato: AndoridManifest. Esso raccolgile le informazioni necessarie al sistema per far girare qualsiasi porzione di codice all’interno dell’applicazione. Tra le altre cose il Manifest si occupa delle seguenti cose:

- Da un nome al package Java dell’applicazione, che è anche un identificatore univoco della stessa;
- Descrive le componenti dell’applicazione, nomina le classi e pubblica le loro “competenze”;
- Determina quali sono i processi che ospiteranno i componenti dell’applicazione;
- Dichiara i permessi ai quali può accedere l’app, e i permessi necessari alle altre app per interagire con essa;
- Dichiara il livello minimo di API Android che l’app richiede;
- Elenca le librerie necessarie all’app per girare.

Per questi motivi, il Manifest può essere visto come il vero cuore di un’applicazione android.

### 3.3.1 Shazarch

L'applicazione creata si basa su 4 cartelle principali:

1. manifests;
2. java;
3. assets;
4. res.

Nella cartella manifests risiede l'AndoridManifest.xml citato poco fa.

La cartella java racchiude tutte le classi e le interfacce utilizzate per creare le componenti. Nello specifico racchiude le classi:

- AutoFitTextureView: ridimensiona la schermata alla dimensione del dispositivo ospitante;
- CFragment: crea l'interfaccia con cui l'utente interagisce;
- Classifier: importa il modello di classificazione tensorflowLite e ne rielabora l'output;
- GPS: riceve le informazioni sulla geolocalizzazione e opera su di esse;
- LogoView: fa visualizzare il risultato della classificazione;
- MainActivity: lancia e gestisce le classi CFragment e GPS;
- Splash: Crea la schermata di apertura dell'applicazione e lancia la MainActivity.

Insieme alle classi risiedono anche le interfacce:

- GPSActivity: necessaria per tutte quelle parti di codice che utilizzano un oggetto di tipo GPS;
- ParentView: necessaria per la classe LogoView.

Nella cartella assets, si inseriscono tutti i file che l'applicazione deve caricare. Qui va inserito, ad esempio, il modello addestrato.

All'interno di res risiedono cartelle riguardanti l'impaginazione, lo stile ed il formato del testo, animazioni ecc...

Per capire quale fosse l'architettura ottimale, sono state realizzate 3 applicazioni:

- Archlite: senza coordinate di geolocalizzazione;
- GPS: con coordinate di geolocalizzazione, considera la distanza tra monumento e osservatore;
- Alarm: con coordinate di geolocalizzazione, considera la presenza del visitatore intorno al monumento.

Introdendo l'acceso al dispositivo GPS, bisogna inserire nell'AndroidManifest la richiesta ai permessi:

```
<uses-permission android:name="android.permission.
                           ACCESS_FINE_LOCATION"/>
<uses-feature android:name="android.hardware.location.gps"/>
```

altrimenti l'applicazione non potrebbe funzionare. La scelta della geolocalizzazione tramite dispositivo GPS è dovuta dal fatto che entrambe le applicazioni devono funzionare senza una connessione dati attiva.

### 3.4 Conclusioni

Attraverso la piattaforma NVIDIA DGX-1, fornita dal dipartimento di Roma3, è stato addestrato il modello di deep learning costruito. Le applicazioni Archlite e GPS sono state testate direttamente sul sito archeologico del Foro Romano. Durante il test sono state fatte all'incirca 400 rilevazioni per 13 siti su 30, una media di 30 foto a reperto, diversificate sia per angolatura che per distanza dal soggetto.

I risultati ottenuti hanno mostrato un'architettura non ancora ottimale: entrambi i modelli hanno un'accuratezza del 52% e una diversificazione tra le due applicazioni da migliorare. Evidentemente la densità di probabilità, calcolata basandosi sulla distanza tra il monumento ed il dispositivo, va aumentata di qualche ordine di grandezza. La GPS provata si basa sulla distribuzione uniforme su un intervallo (0,d), dove d è la distanza, che è lineare. In GPS, l'etichetta predetta ha una probabilità maggiore di "match" con l'etichetta effettiva, in confronto ad Archlite, quando ci si trova in prossimità del sito, come mostrato nella figura: 3.1.

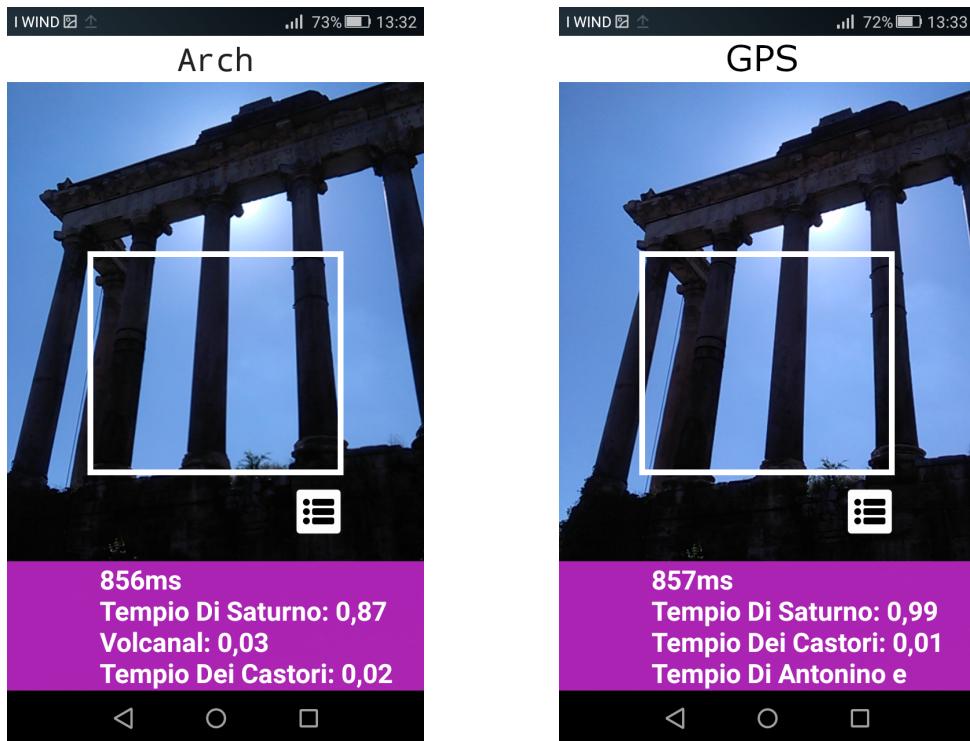


Figura 3.1: Schermata catturata ai piedi del Tempio di Saturno, a sinistra: il risultato dell'app ArchLite, a destra: il risultato dell'app GPS

Viceversa la probabilità è più bassa quando la foto risulta “panoramica”, ovvero quando il sito di interesse è lontano dall’obiettivo della fotocamera. Si è osservato che nei casi in cui si è vicini al sito, se la predizione è corretta, la probabilità registrata è sempre molto alta, mentre quando la previsione è errata, l’app GPS registra una probabilità più bassa, si veda la figura 3.2. Denotando un comportamento coerente con la “filosofia” con cui è stato costruito.



Figura 3.2: Schermata catturata della Basilica Emilia, erroneamente classificata. A sinistra la predizione dell'app ArchLite, a destra l'app GPS.

Si è riscontrato un fenomeno di overfitting: l’etichetta corrispondente alla “Domus Augustana” è spesso tra i risultati più probabili, se non il più probabile. Il fenomeno si è verificato in quanto, l’area della Domus Augustana è molto vasta, perciò il numero di foto necessarie al riconoscimento sono più numerose. La rete quindi è stata allenata troppo con queste immagini e tende a riconoscere soprattutto loro. Proprio a causa di questo comportamento, i risultati sono stati inficiati dalla predominanza di questa etichetta. Nei casi in cui il valore predetto non corrisponde a quello effettivo, la Domus Augustana compare come risultato più probabile nel 50% dei casi. Come nella figura (3.2) dove, invece di classificare l’immagine come la Basilica Emilia, l’errore di predizione è ricaduto sulla Domus Augustana.

Quando il modello viene testato, tramite il suo insieme di validazione, su DGX l’accuratezza è molto elevata, misurando in alcuni casi anche il 100% di accuratezza. Mentre durante il test ai Fori l’accuratezza è molto bassa: 53% per entrambe le applicazioni ma, nell’analizzare i valori di probabilità registrati, GPS sembra essere (seppur di poco) più attendibile, anche se lontana dall’essere ottimale, ma ci sono margini di miglioramento possibili.

# Bibliografia

- [1] Arthur Samuel. [https://www.ibm.com/developerworks/community/blogs/jfp/entry/What\\_Is\\_Machine\\_Learning?lang=en](https://www.ibm.com/developerworks/community/blogs/jfp/entry/What_Is_Machine_Learning?lang=en), 1959.
- [2] Shehzad Noor Taus Priyo. <https://towardsdatascience.com/intro-to-deep-learning-d5caceedcf85>, 2017.
- [3] David H. Wolper and William G. Macready. No free lunch theorems for optimization, 1996.
- [4] Roberto Ferretti. Appunti del corso di analisi numerica.
- [5] Ryan Tibshirani. Error and validation.
- [6] Wikipedia. Clustering. <https://it.wikipedia.org/wiki/Clustering>, 2018.
- [7] Frank L Lewis. *Reinforcement learning and approximate dynamic programming for feedback control*. Wiley, 2013.
- [8] Sito Ufficiale dell'Università di Princeton. What is wordnet? <https://wordnet.princeton.edu/>.
- [9] Andreetto Howard Menglong Chen Kalenichenko Wang Weyand and Adam. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, 2017.
- [10] Marcus D.Bloice. <https://augmentor.readthedocs.io/en/master/>, 2016.
- [11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- [12] K.P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive computation and machine learning. MIT Press, 2012.

- [13] Kai Li Jia Deng Wei Dong Richard Socher, Li-Jia Li and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. *Dept. of Computer Science, Princeton University, USA*, 2010.
- [14] TensorFlow. <https://www.tensorflow.org>.
- [15] Giancarlo Zacccone. *Getting Started with TensorFlow*. Packt Publishing Ltd, 2016.
- [16] Rajalingappaa Shanmugamani. *Deep learning for computer vision*. Packt Publishing Ltd, 2018.