

ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA  
SEDE DI CESENA

---

Scuola di Ingegneria ed Architettura  
Corso di Laurea in Ingegneria Elettronica per l'Energia e  
l'Informazione

IMPLEMENTAZIONE DI UNA  
RETE NEURALE  
CONVOLUZIONALE CON  
DATASET CUSTOM PER LA  
CLASSIFICAZIONE DI VEICOLI

Elaborato in  
Laboratorio di Programmazione di Reti e Dispositivi Mobili

*Tesi di Laurea di:*  
ROBERTO NERI

*Relatore:*  
Prof. Ing.  
ENRICO PAOLINI  
*Correlatore:*  
Prof. Ing.  
GIANNI PASOLINI

---

SESSIONE II  
ANNO ACCADEMICO 2019-2020



# PAROLE CHIAVE

CNN

Tensorflow

Keras

Dataset

Python



# Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Machine Learning e Reti Neurali</b>	<b>3</b>
1.1 Neuroni Artificiali . . . . .	3
1.1.1 Perceptron . . . . .	4
1.1.2 Adaline . . . . .	4
1.1.3 Logistic Regression . . . . .	6
1.1.4 One vs All . . . . .	8
1.1.5 Adam . . . . .	8
1.2 Reti Neurali . . . . .	8
1.2.1 Forward Propagation . . . . .	9
<b>2 Reti Neurali Convolutionali</b>	<b>13</b>
2.1 Funzionamento Generale . . . . .	13
2.2 Convoluzione e Convoluzione in 2D . . . . .	14
2.2.1 Convoluzione . . . . .	14
2.2.2 Padding . . . . .	15
2.2.3 Convoluzione 2D . . . . .	17
2.2.4 Strati di subsampling . . . . .	17
2.3 Implementazione di una CNN . . . . .	18
2.3.1 Dropout . . . . .	19
2.3.2 Loss functions . . . . .	19
2.3.3 Attivazioni . . . . .	19
<b>3 Python, Tensorflow e Keras</b>	<b>21</b>
3.1 Python . . . . .	21
3.2 Tensorflow e Keras . . . . .	21
3.2.1 Utilizzo delle GPU . . . . .	22
3.3 Ubuntu 20.04.1 LTS . . . . .	22
3.4 Labelbinarizer . . . . .	23

---

<b>4</b>	<b>Implementazione Software e Prove Sperimentali</b>	<b>25</b>
4.1	Preparazione dell'ambiente di lavoro . . . . .	25
4.2	Algoritmo VGG16 . . . . .	26
4.3	Creazione Dataset . . . . .	27
4.3.1	Train Test Split . . . . .	28
4.3.2	Utilizzo di un Dataset Pre-Costruito . . . . .	29
4.4	Caricare le immagini . . . . .	29
4.5	Grafici . . . . .	30
4.6	Test Svolti . . . . .	31
4.6.1	Algoritmo didattico . . . . .	31
4.6.2	Algoritmi Custom . . . . .	32
4.6.3	Veichle Detection ConvNet [17] . . . . .	35
4.6.4	Differenze di Prestazioni tra Dataset Custom e Pre-Costruito . . . . .	36
4.6.5	Rimozione di ImageDataGenerator . . . . .	37
4.6.6	Test con immagini . . . . .	39
	<b>Conclusioni</b>	<b>41</b>
	<b>Codice</b>	<b>43</b>
	<b>Elenco Figure</b>	<b>47</b>
	<b>Bibliografia</b>	<b>49</b>

---

# Introduzione

Lo scopo di questo elaborato è la creazione di un algoritmo per la classificazione di veicoli. Per questo si è affrontato il problema della classificazione non binaria di immagini appartenenti ad un *Dataset* non preesistente tramite *CNN* (Convolutional Neural Network o Rete Neurale Convolutionale): una tecnica di *Machine Learning* considerata, attualmente, lo “stato dell’arte” per quanto riguarda gli algoritmi per la classificazione di immagini.

Con *Machine Learning* si intende un insieme di algoritmi che abilitano la possibilità di “apprendere” ai codici. Dopo un addestramento sarà possibile utilizzare questo codice per effettuare previsioni o classificazioni relative all’ambito (ed al tipo di dato) del *Dataset* utilizzato.

Il *Dataset* è un insieme di dati (che possono essere sotto forma di vettori, immagini, suoni ecc.) suddiviso in *classi* con ogni elemento associato ad una ed una sola di queste. Vengono utilizzati in fase di addestramento ed è estremamente importante che questi sia costruito in maniera corretta.



# Capitolo 1

## Machine Learning e Reti Neurali

In questo capitolo viene fatta un'introduzione al *Machine Learning*:

- Costrutti principali;
- Algoritmi di classificazione binaria;
- Algoritmi di classificazione OvA (One versus All).

Successivamente vengono introdotte le Reti Neurali (NN): modelli di *Machine Learning* che permettono di elaborare dati più complessi come ad esempio immagini.

### 1.1 Neuroni Artificiali

McCulloch e Pitts nel loro studio definiscono i neuroni come delle semplici porte logiche con un output binario: arrivano dei segnali ai dendriti che vengono poi integrati nel corpo della cella e, se i segnali accumulati superano una certa soglia, viene trasmesso sul neurite un segnale di uscita [2]. Pochi anni dopo Frank Rosenbalt pubblicò il primo articolo in cui parlava del *Perceptron* basato sul modello MCP del neurone. Rosenblatt propose un algoritmo che avrebbe imparato in automatico i pesi ottimali dei coefficienti che sarebbero poi stati moltiplicati con i dati in ingresso per capire se un neurone trasmette un segnale o meno. Questo fu il primo algoritmo di classificazione binaria [3]. I *Neuroni Artificiali* come ad esempio il *Perceptron* o l'*Adaline* hanno degli "iperparametri", sono due quelli principali  $\eta$  detto *Learning Rate* e il numero di iterazioni (`n.iter`). È fondamentale regolare questi parametri in

maniera corretta: se, per esempio, si scegliesse un  $\eta$  troppo piccolo, il codice avrebbe un tempo di esecuzione decisamente maggiore rispetto ad uno con  $\eta$  più grande, tuttavia un  $\eta$  troppo grande potrebbe portare ad un loop infinito. Allo stesso modo il `n_iter` va regolato in base alle prestazioni dell'algoritmo; essendo questo il numero di volte in cui viene addestrato non ha senso prendere un numero troppo grande poiché si raggiungerebbe un punto di convergenza (come si vedrà più avanti la funzione di apprendimento sarà logaritmica) prima della fine di tutti i calcoli, mentre averlo troppo piccolo rischierebbe di impedire al codice di arrivare al punto di convergenza.

### 1.1.1 Perceptron

Formalmente, è possibile pensare ad un *Neurone Artificiale* come ad un classificatore binario dove le classi sono rappresentate da 1 (classe positiva) o -1 (classe negativa). È ora possibile definire una funzione decisionale  $\Phi(z)$  che prende in ingresso una combinazione lineare del vettore degli input,  $\mathbf{x}$ , e un vettore dei pesi,  $\mathbf{w}$ , con  $z$  la rete di ingresso  $z = w_1x_1 + w_2x_2 + \dots + w_mx_m$ . Ora se l'ingresso di un certo esempio,  $\mathbf{x}^{(i)}$ , risulta più grande di una certa soglia,  $\theta$ , la classe predetta sarà 1, -1 altrimenti; questa viene chiamata funzione a gradino ed è mostrata in Figura 1.1. Si possono ora definire  $w_0 = -\theta$  e  $x_0 = 1$  in modo da poter riscrivere l'equazione in maniera più compatta:

$$z = w_0x_0 + w_1x_1 + \dots + w_mx_m = \mathbf{w}^T \mathbf{x}$$

Si ottiene quindi:

$$\Phi(z) = \begin{cases} 1 & z \geq 0, \\ -1 & \text{altrimenti.} \end{cases}$$

La soglia negativa ( $w_0 = -\theta$ ), in letteratura, viene spesso chiamata “*bias unit*” ed è associata alla sola  $x_0 = 1$  detta attivazione. Il vettore dei pesi  $\mathbf{w}$  viene aggiornato ad ogni iterazione seguendo la logica:

$$\mathbf{w} := \mathbf{w} + \Delta \mathbf{w}$$

dove

$$\Delta \mathbf{w} = -\eta \nabla J(\mathbf{w}).$$

### 1.1.2 Adaline

*Adaline* (ADaptive LInear NEuron) viene considerato come il primo vero miglioramento del *Perceptron*, la principale differenza tra questi è che *Adaline*

---

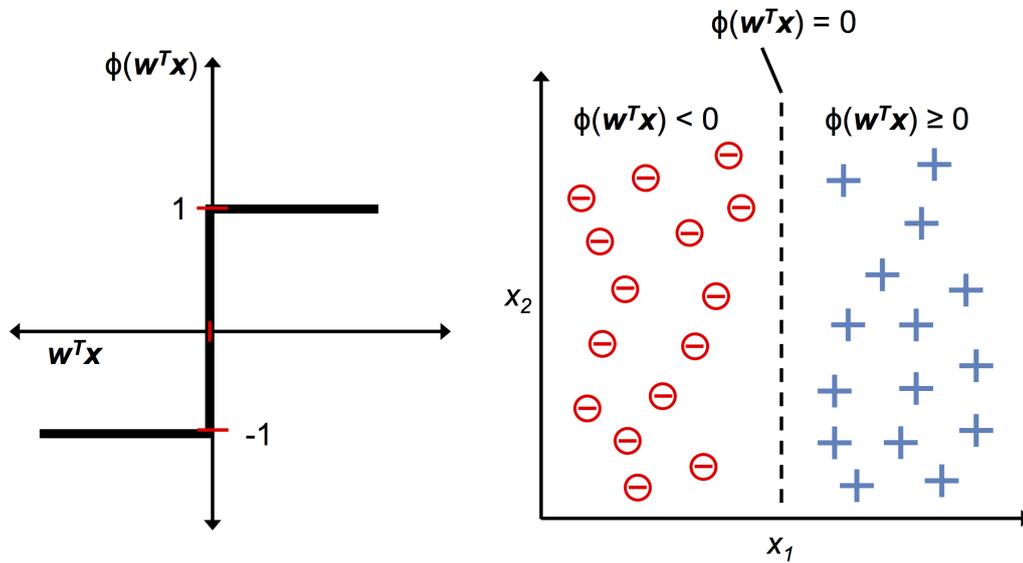


Figura 1.1: Output binario di  $\theta(z)$  [1].

utilizza una funzione di attivazione lineare per regolare il vettore dei pesi ( $\mathbf{w}$ ) al posto della funzione a gradino. Viene comunque utilizzata una funzione a soglia per l'output [4]. Con *Stochastic Gradient Descent* si intende un algoritmo molto utilizzato per quanto riguarda l'apprendimento del singolo neurone. Come si evince dal nome si tratta di trovare il minimo della funzione

$$J(\mathbf{w}) = \frac{1}{2} \sum_i (y^{(i)} - \Phi(z^{(i)}))^2$$

dove

- $J(\mathbf{w})$  è la funzione da minimizzare;
- $y^{(i)}$  è il valore desiderato in uscita, o l'uscita reale;
- $\Phi(z^{(i)})$  è la funzione in uscita dall'ottimizzatore (*Adam, Adeline, ecc.*).

Il metodo migliore per trovare il minimo di una funzione è attraverso il gradiente, ossia effettuando derivate successive fino a trovare il punto che lo annulli. Teoricamente anche i massimi delle funzioni hanno gradiente nullo ma le funzioni in questione non hanno punti di massimo assoluti, è pertanto possibile assumere che l'annullamento del gradiente corrisponda sempre ad un minimo. Questo processo è "semplice" per quanto riguarda un singolo ottimizzatore, poiché la funzione, tendenzialmente avrà un solo minimo

globale. Purtroppo le cose si complicano in ambito di *Reti Neurali* dove il maggior numero di ottimizzatori fa sì che la funzione si complichino notevolmente portando alla creazione di vari minimi locali. In questi punti, se  $\eta$  è troppo piccolo, il codice potrebbe fermarsi poiché troverebbe un gradiente a 0 e non riuscirebbe a uscire dal minimo locale, a causa del differenziale troppo piccolo. Tuttavia, utilizzando un valore troppo grande di  $\eta$  l'algoritmo rischia di entrare in un loop infinito poiché effettuerebbe salti troppo grandi e non arriverebbe mai al minimo. In Figura 1.2 si può osservare la differenza

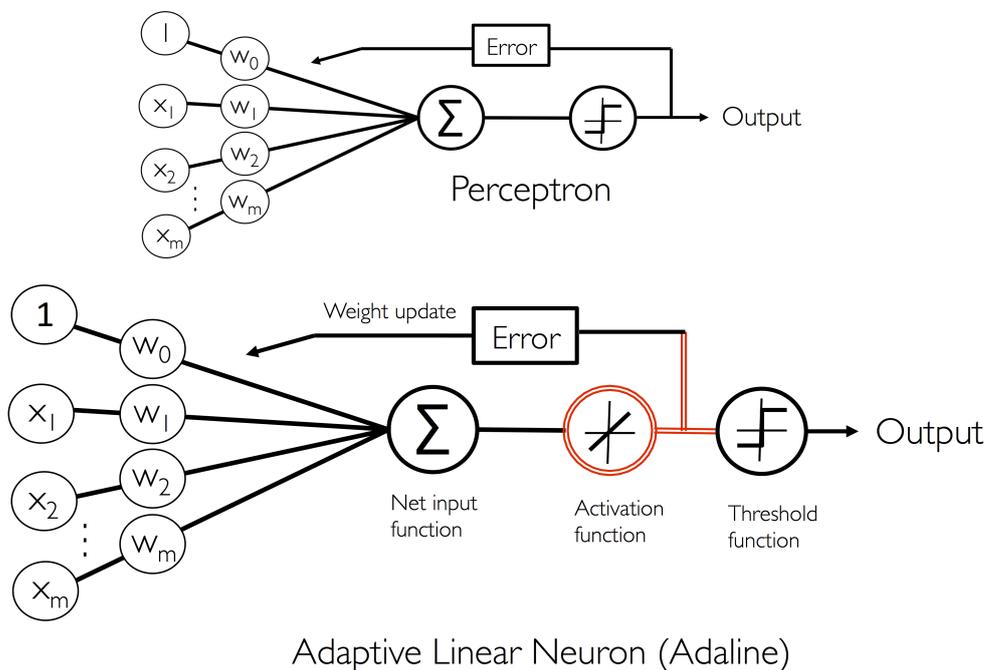


Figura 1.2: Differenze tra *Perceptron* e *Adaline* [1].

tra Perceptron e Adaline a livello logico.

### 1.1.3 Logistic Regression

Il principale problema dello *Stochastic Gradient Descent* è che la funzione non converge mai se le classi non sono perfettamente separabili. Prima di parlare di *Logistic regression* è importante parlare del concetto di possibilità che accada un evento  $p$ . Questa possibilità è descritta dalla funzione

$$\frac{p}{1-p}$$

dove  $p$  rappresenta la probabilità che si verifichi quel determinato evento. Si definisce funzione *logit* il logaritmo naturale della possibilità:

$$\text{logit}(p) = \ln \frac{p}{1-p}$$

Questa operazione ha come obiettivo quello di trasformare un valore compreso tra 0 e 1 in un valore da  $-\infty$  a  $+\infty$ .

$$\text{logit}(p(y = 1|x)) = \mathbf{w}^T \mathbf{x}$$

dove  $p(y = z|x)$  è la probabilità condizionata che un certo esempio appartenga alla classe  $z$  conoscendo le sue caratteristiche  $\mathbf{x}$ . Tuttavia lo scopo dell'algoritmo è capire quale sia la probabilità che un determinato elemento faccia parte di una certa classe, che è l'operazione opposta. Questa funzione si chiama *Sigmoid Function* ed è illustrata in Figura 1.3.

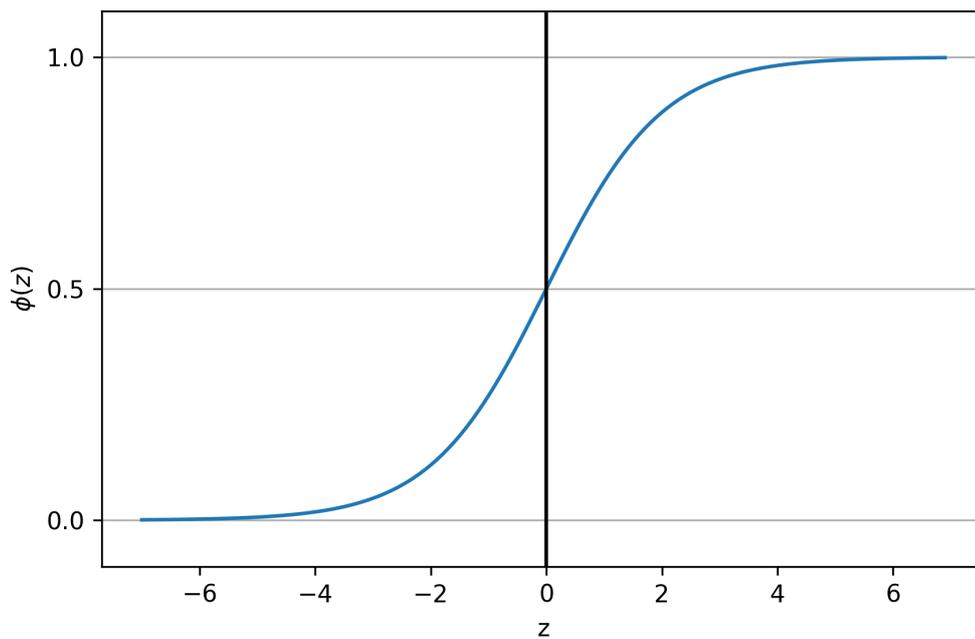


Figura 1.3: Sigmoid Function [1].

$$\Phi(z) = \begin{cases} 1 & z \rightarrow +\infty \\ 0 & z \rightarrow -\infty \end{cases}$$

È, in pratica, una funzione che ha come ingresso dei valori in tutto l'insieme dei numeri reali e riporta in uscita un valore compreso tra 0 e 1.

---

### 1.1.4 One vs All

Questo non è un neurone specifico quanto un insieme di algoritmi: OvA (One versus All) si riferisce all'insieme di algoritmi di classificazione non binaria che sfruttano una serie di classificazioni binarie. Prendendo come esempio 3 classi (potrebbero essere anche 10,20,100,ecc.) l'algoritmo controlla se l'ingresso appartiene alla classe 1 (verranno scritte le classi 1-2-3 in sequenza per comodità ma non è detto che questo sia l'ordine con cui lavora l'algoritmo), con un esito positivo si ha un uscita ma con esito negativo si andrà a fare lo stesso controllo sulla classe 2 ed, eventualmente, anche nella classe 3. Sono algoritmi piuttosto semplici in quanto l'ingresso viene trasferito identico nei vari casi e si tratta sostanzialmente di 3 (o più) classificatori binari ognuno dei quali attivato da un esito "negativo" del precedente.

### 1.1.5 Adam

*Adam* è un ottimizzatore (simile ad *Adaline* e *Perceptron*) che si basa su un algoritmo chiamato "Stochastic Gradient Descend" e sull'unione di due ottimizzatori preesistenti: *AdaGrad* (Adaptive Gradient) ed *RMSProp* (Root Mean Square PROPagation) combinandone i pregi [5][6]. Ad oggi *Adam* viene considerato lo stato dell'arte per quanto riguarda i neuroni nella *Computer Vision* e nel riconoscimento di linguaggi "naturali". *Adam* è l'ottimizzatore che è stato usato nelle prove effettuate.

## 1.2 Reti Neurali

Il *Deep Learning* è uno degli argomenti attualmente più discussi in materia di *Machine Learning* in quanto tratta i metodi per addestrare le *Reti Neurali*. Le *Deep Neural Networks (DNNs)* sono appunto *Reti Neurali* che utilizzano questi metodi per massimizzare le prestazioni. Esistono periodi storici (principalmente 2) in cui i ricercatori hanno perso interesse nel *Machine Learning* a causa del fatto che, dopo la *Perceptron Rule* (di cui si parla nel paragrafo 1.1.1), non si sono trovati, per lunghi periodi, algoritmi validi per l'addestramento delle *Reti Neurali* [7]. Quando si parla di *Reti Neurali* si può pensare ad un neurone con più vettori dei pesi. Ogni vettore dei pesi costituisce uno strato chiamato *layer*. Si possono dividere i layer in 3 gruppi:

- Input layer, o strato di ingresso;
  - Hidden layers, o strati nascosti;
  - Output layer, o strato di uscita.
-

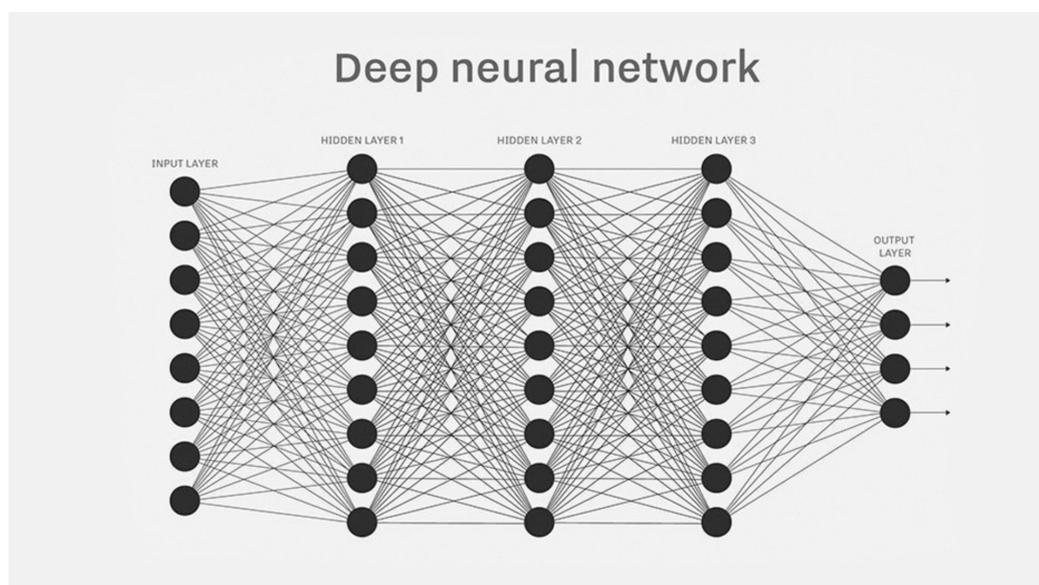


Figura 1.4: Esempio di *Rete Neurale* con più “*Hidden Layers*”

Mentre si ha un solo strato di ingresso ed un solo strato di uscita si possono avere più strati nascosti <sup>1</sup>.

### 1.2.1 Forward Propagation

Il processo di apprendimento nelle *Reti Neurali* si può dividere in 3 fasi principali:

1. Iniziando dall’input layer, si fa passare i pattern dei dati di addestramento attraverso la rete per ottenere un output;
2. In base all’uscita si calcola l’errore che si vuole minimizzare utilizzando una funzione che verrà descritta successivamente;
3. L’errore viene riportato in ingresso, si trova la sua derivata rispetto a ciascun peso nella rete e si aggiorna il modello [1].

Dopo aver ripetuto questo processo più volte, è possibile assegnare delle classi alle uscite. La rappresentazione più utilizzata per le classi è il cosiddetto metodo “one hot”. Questo metodo consiste di un vettore (riga o colonna è

---

<sup>1</sup>Lo strato di ingresso corrisponde, come numero di elementi, al numero degli elementi in ingresso con la possibilità di aggiungere la bias unit, quindi un ingresso a 1. Le bias unit possono essere aggiunte ad ogni strato eccezion fatta per quello in uscita che deve avere un numero di elementi pare al numero delle possibili uscite)

indifferente) di lunghezza pari al numero delle classi, ad ogni elemento viene assegnato uno 0 eccezion fatta per l'elemento corrispondente alla posizione della classe (Es. ipotizzando di avere 3 classi: cani, gatti e criceti; alla classe cani (0) corrisponderà un vettore con l'elemento in posizione 0 a 1 e il resto tutti 0). Per prima cosa è necessario ricavare il valore dell'attivazione appartenente allo strato nascosto [1]:

$$z_1^{(h)} = a_0^{(in)} w_{0,1}^{(h)} + a_1^{(in)} w_{1,1}^{(h)} + \dots + a_m^{(in)} w_{m,1}^{(h)}$$

$$a_1^{(h)} = \Phi(z_1^{(h)})$$

Dove  $z_1^{(h)}$  è l'ingresso della rete e  $\Phi(z_1^{(h)})$  è la funzione di attivazione, che deve essere derivabile per poter sfruttare un algoritmo a gradiente per trovare il costo minimo. Per risolvere problemi complessi, come ad esempio la classificazione di immagini o il riconoscimento vocale, è consigliabile usare una funzione di attivazione non lineare. Il *sigmoide* (Figura 1.3) risulta la funzione ottimale per questo tipo di applicazioni. Questo tipo di *Reti Neurali* viene chiamato "*feedforward*", questo termine sta a significare che ogni strato funge da input per quello successivo, senza la creazione di cicli (come invece accade nelle *Recurrent Neural Network (RNNs)* [8] che non verranno trattate in questo elaborato). Queste reti vengono chiamate MLP (*MultiLayer Perceptron*) nonostante usino un sigmoide come funzione di attivazione e non *Perceptron*, si può tuttavia pensare alle MLP come a delle entità di tipo *Logistic Regression* che danno in uscita valori compresi tra 0 e 1. Per facilitare la traduzione a codice (verrà usato il linguaggio *Python* che possiede una libreria (*NumPy*) in grado di lavorare con vettori e matrici) del seguente algoritmo verranno ora riportate delle elaborazioni della funzione vista poco sopra [1]:

$$\mathbf{z}^{(h)} = \mathbf{a}^{(in)} \mathbf{W}^{(h)}$$

$$\mathbf{a}^{(h)} = \Phi(\mathbf{z}^{(h)})$$

Dove:

- L'apice  $h$  sta ad indicare l'appartenenza dell'elemento agli strati nascosti;
  - L'apice  $in$  indica l'appartenenza allo strato di ingresso;
  - $\mathbf{a}^{(in)}$  è il vettore dei pesi (con dimensionalità  $1 \times m$ ) in ingresso il cui numero corrisponde a quello degli ingressi +1 (attivazione);
  - $\mathbf{W}^{(h)}$  è una matrice ( $d \times m$ ) che ha righe pari al numero di ingressi e colonne pari al numero di strati nascosti;
-

- $\mathbf{z}^{(h)}$  è un vettore ( $1 \times d$ ) utilizzato per calcolare l'attivazione dello strato nascosto.

I calcoli precedenti prendono in considerazione dei vettori in quanto sono calcoli effettuati per ogni elemento del *Dataset*, volendo generalizzare ad  $n$  elementi si ottiene [1]:

$$\begin{aligned}\mathbf{Z}^{(h)} &= \mathbf{A}^{(in)}\mathbf{W}^{(h)} \\ \mathbf{A}^{(h)} &= \Phi(\mathbf{Z}^{(h)})\end{aligned}$$

Dove quelli che precedentemente erano vettori del tipo  $1 \times m$  o  $1 \times d$  divengono matrici  $n \times m$  o  $n \times d$  rispettivamente. Dopodiché si moltiplica il risultato ottenuto ( $\mathbf{A}^{(h)}$ ) con una matrice ( $\mathbf{W}^{(out)}$ ) di dimensione  $d \times t$  (con  $t$  il numero di uscite) per ottenere la funzione di uscita globale [1]:

$$\mathbf{Z}^{(out)} = \mathbf{A}^{(h)}\mathbf{W}^{(out)}$$

Anche in questo caso, infine, per ottenere la funzione di attivazione globale della MLP si applica di nuovo la funzione *sigmoide* in uscita [1]:

$$\mathbf{A}^{(out)} = \Phi(\mathbf{Z}^{(out)})$$

con

$$\mathbf{A}^{(out)} \in \mathbb{R}^{n \times t}$$

Questo modello è la base delle *Reti Neurali Convolutionali (CNN)* un modello di Rete Neurale considerato il migliore per la classificazione di immagini.

---



## Capitolo 2

# Reti Neurali Convolutionali

Le *Reti Neurali Convolutionali (CNN)* nascono nel 1990 dalla ricerca di Yann LeCun insieme al suo team basandosi sul funzionamento della corteccia visiva del cervello umano [9][10]. Grazie alle ottime prestazioni che si sono riuscite a ricavare soprattutto in ambito del riconoscimento di immagini, ancora oggi le *CNN* sono considerate lo “stato dell’arte” per quanto riguarda riconoscimento di pattern ed immagini. Yann LeCun ha ricevuto, nel 2019, il premio Turing per il suo lavoro.

### 2.1 Funzionamento Generale

Un aspetto chiave di ogni algoritmo di *Machine Learning* è quello di riuscire ad estrarre i tratti più importanti dai dati in ingresso. Le *CNN* sono in grado di capire in maniera automatica i tratti più rilevanti e di apprendere i pattern; per questo motivo di norma gli strati delle *CNN* vengono considerati come dei blocchi per rilevare i tratti: i primi strati (quelli subito dopo lo strato di ingresso) sono considerati “low-level features extractors”, mentre gli ultimi strati (di solito completamente connessi come quelli delle *Reti Neurali* non convolutive discusse alla Sezione 1.2) sono considerati “high-level features extractors”. Nelle immagini le “low-level features” sono ad esempio i bordi o i blob (in letteratura vengono definiti blob alcuni tipi di blocchi elementari) che vengono rielaborate per formare delle “high-level features” le quali, una volta date in pasto agli ultimi strati della rete, saranno in grado di creare, ad esempio, i contorni di case, cani, gatti o qualsiasi cosa fosse presente nell’immagine originale. Le *CNN* elaborano delle “mappe dei tratti” dove ogni elemento corrisponde a dei pixel nell’immagine originale. Per ottenere questo risultato è necessario effettuare un’operazione detta *convoluzione*. Questa operazione è piuttosto complicata e verrà discussa in

maniera più approfondita nella Sezione 2.2. Vi sono due concetti alla base del funzionamento delle *CNN*:

- *Sparse Connectivity* (connettività scarsa): un singolo elemento nella mappa dei tratti è connesso solo a piccoli gruppi di pixel;
- *Parameter-sharing* (condivisione dei parametri): gli stessi pesi vengono utilizzati per diversi gruppi di pixel dell'immagine principale.

Sostituire uno strato di tipo MLP con uno convolutivo porta ad una drastica diminuzione del numero di parametri (o pesi) all'interno della rete e, quindi, ad un miglioramento nella capacità di estrarre tratti rilevanti (nel riconoscimento di immagini è corretto assumere che pixel vicini siano più rilevanti tra di loro rispetto a pixel lontani l'uno dall'altro). Le *CNN* sono formate da vari strati convolutivi e strati di sottocampionamento (subsampling) seguiti da strati completamente connessi (praticamente delle MLP) alla fine. Gli strati convolutivi e quelli completamente connessi hanno pesi e attivazioni (già discussi nella Sezione 1.1.1) come quelli presenti nei *Perceptron* e questi vengono ottimizzati per l'apprendimento. Gli strati di sottocampionamento (pooling layers) invece no.

## 2.2 Convoluzione e Convoluzione in 2D

In questo paragrafo verrà brevemente rivista la convoluzione di vettori e di matrici da un punto di vista prettamente matematico. È importante far notare che queste operazioni vengono svolte da istruzioni all'interno delle librerie utilizzate perciò non verranno esplicitate nel codice. Nella Sezione 2.2.1 si parlerà della convoluzione di due vettori mentre, nella Sezione 2.2.3, si parlerà della convoluzione di matrici.

### 2.2.1 Convoluzione

Si definisce prodotto di convoluzione il prodotto  $\mathbf{y} = \mathbf{x} * \mathbf{w}$  dove  $\mathbf{x}$  rappresenta il vettore in ingresso, e  $\mathbf{w}$  rappresenta la funzione detta *filtro* (o *kernel* per la convoluzione di matrici):

$$\mathbf{y} = \mathbf{x} * \mathbf{w} \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i - k]w[k]$$

Un primo problema di questa definizione sta nell'indice  $k$ : in applicazioni reali non è possibile computare infiniti valori (si entrerebbe in un loop senza

---

uscita) e, inoltre, i dati in ingresso sono anch'essi sempre finiti. Nella realtà l'indice  $k$  varia tra due valori finiti, compresi tra 0 e un valore più grande della dimensione del vettore di ingresso, e vengono aggiunti degli 0 al nostro vettore di interesse prima e dopo (*zero padding*) come visto in Figura 2.1. In

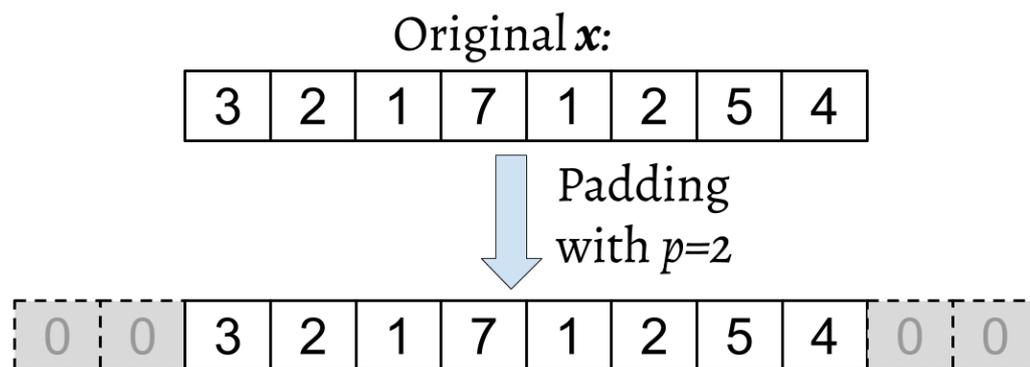


Figura 2.1: Esempio di zero padding con l'aggiunta di 4 "0"[1].

questo modo si ottiene un vettore  $\mathbf{x}^p$  di lunghezza  $n + 2p$ . È ora opportuno aggiornare il prodotto di convoluzione scritto in precedenza:

$$y[i] = \sum_{k=0}^{k=m-1} \mathbf{x}^p[i + m - k]w[k][1]$$

Dove  $m$  è la lunghezza del vettore  $\mathbf{w}$ . Come si può notare dalle equazioni sopra riportate gli indici dei due vettori scorrono in direzione opposta ( $\mathbf{x}^p$  ha  $-k$  mentre  $\mathbf{w}$  ha  $k$ ). Invertendo il vettore  $\mathbf{w}$  (nell'ipotesi che  $m \leq n$  con  $n$  la dimensione di  $\mathbf{x}^p$ ) si ottiene il vettore  $\mathbf{w}^r$ , in questo modo gli indici nell'equazione sopra riportata scorrono nella stessa direzione. A questo punto si può eseguire il prodotto puntuale di  $\mathbf{w}^r$  con una "patch" (frazione) di  $\mathbf{x}^p[i : i + m]$  e ripetere con diverse patch finché il prodotto non è terminato. Altro parametro molto importante per il prodotto di convoluzione è lo spostamento del filtro per ogni calcolo (detto *stride* nelle applicazioni di *CNN*): ad ogni passaggio, infatti, il filtro viene spostato di un numero di posizioni ( $z$ ) uguale allo scorrimento. Lo *stride* è un iperparametro aggiuntivo proprio delle *CNN*.

### 2.2.2 Padding

Nella Sezione 2.2 si è visto come il *padding*, in teoria, possa avere  $p \geq 0$ . Tuttavia in base alla scelta di  $p$  alcuni elementi del vettore  $\mathbf{x}^p$  potrebbero venire trattati in maniera diversa (ad esempio con  $n = 5$  ed  $m = 3$  scegliendo

$p = 0$  il primo elemento di  $\mathbf{x}$ ,  $x[0]$ , verrebbe utilizzato nel solo calcolo di  $y[0]$ , mentre  $x[1]$  verrebbe utilizzato nel calcolo di due elementi [1]). Esistono tre tipologie principali di *padding* (mostrate in Figura 2.2):

- “Full padding”: con questa tipologia si ha  $p = m - 1$ . Tuttavia nel full padding la dimensione di  $\mathbf{y}$  risulta maggiore di quella di  $\mathbf{x}$  perciò non viene usato in applicazioni di *CNN*;
- “Same padding”: nel same padding si fa in modo che  $\mathbf{y}$  e  $\mathbf{x}$  abbiano la stessa dimensione. La macchina calcola in automatico  $p$  per riuscire a rispettare questa condizione;
- “Valid padding ”: nel valid padding  $p = 0$ .

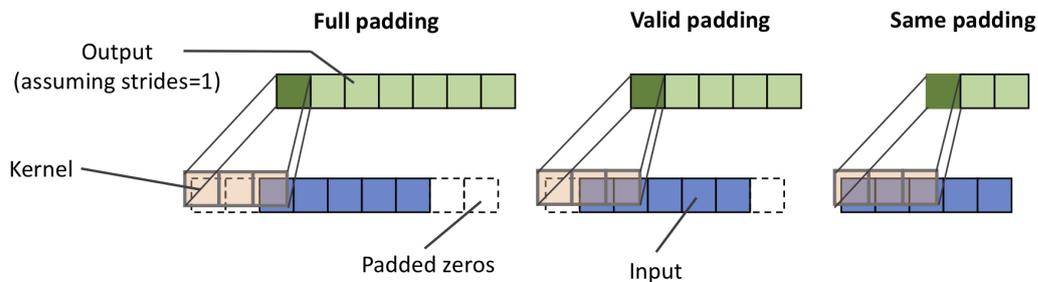


Figura 2.2: Output con i 3 tipi di padding [1].

Nelle *CNN* si utilizza quasi sempre *same padding* per poter preservare la dimensione dell'ingresso originale (che sia un vettore o una matrice) in modo da avere in output la stessa dimensione dell'immagine in ingresso. La dimensione del risultato di una convoluzione è data da:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

dove

- $n$  è la dimensione di  $\mathbf{x}$ ;
- $p$ , come visto in precedenza, è il padding;
- $m$  è la dimensione di  $\mathbf{w}$ ;
- $s$  è lo stride.

Da qui è semplice ricavare  $p$  (*same padding*) e/o  $s$  per avere  $o$  del valore desiderato.

### 2.2.3 Convoluzione 2D

Gli stessi ragionamenti visti nella Sezione 2.2.2 possono essere applicati a delle matrici. Siano  $\mathbf{X}_{n_1 \times n_2}$  e  $\mathbf{W}_{m_1 \times m_2}$  due matrici tali per cui  $m_1 \leq n_1$  e  $m_2 \leq n_2$  allora sarà possibile ottenere una matrice  $\mathbf{Y}$  tale per cui:

$$\mathbf{Y} = \mathbf{X} * \mathbf{W} \rightarrow Y[i, j] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} X[i - k_1, j - k_2] W[k_1, k_2]$$

Con  $\mathbf{Y}$  chiamato prodotto di convoluzione tra matrici (o *Convoluzione 2D*). È importante notare come, togliendo una dimensione dalla equazione precedente, si ottiene esattamente quella descritta nella Sezione 2.2.2, questo fa capire come tutte le proprietà e le tecniche sopra citate (*padding, stride* e rotazioni) sono applicabili anche alla *Convoluzione 2D* purché vengano applicate indipendentemente ad entrambe le dimensioni.

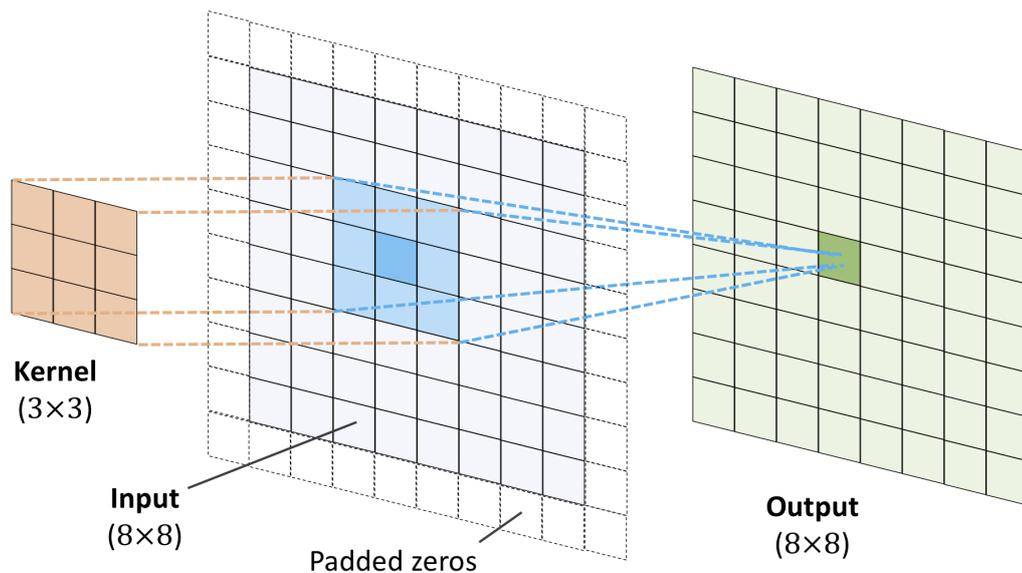


Figura 2.3: Convoluzione 2D con same padding [1].

### 2.2.4 Strati di subsampling

Esistono due tipi di *Subsampling layers*:

- Max-Pooling;
- Mean-Pooling.

Il codice determina il numero di pixel vicini (*neighbors*) per ottenere delle matrici del tipo  $\mathbf{P}_{n_1 \times n_2}$  sulle quali verrà poi effettuata una delle due operazioni. Con il *Max-Pooling* si prenderà, da ogni sottomatrice  $\mathbf{P}$ , il valore più grande del *neighborhood*, nel caso di *Mean-Pooling*, invece, verrà calcolato il valore medio di ogni *neighborhood*. Il *Max-Pooling* introduce invarianza, questo vuol dire che piccoli cambiamenti non cambieranno il risultato finale, perciò aggiunge robustezza ai dati. Inoltre utilizzare il *Pooling* riduce le dimensioni dei tratti fondamentali, rendendo la computazione più efficiente.

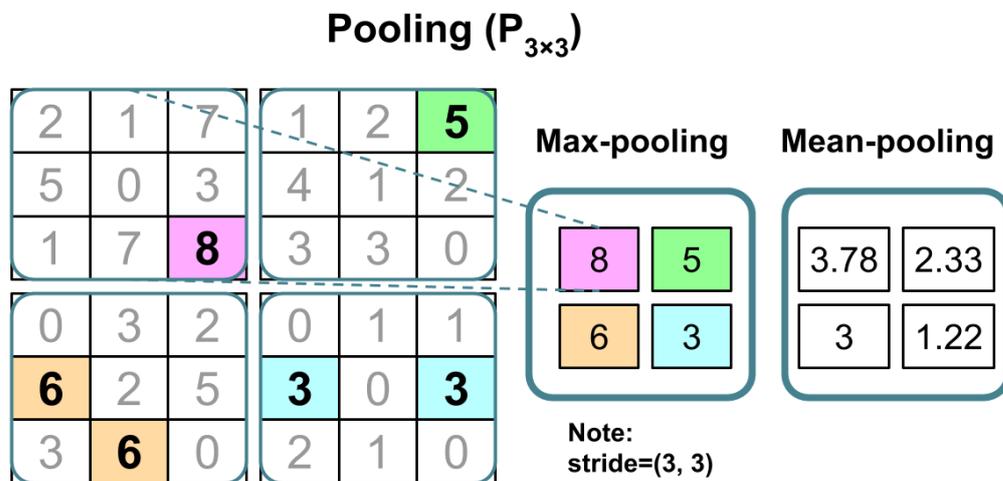


Figura 2.4: Pooling [1].

## 2.3 Implementazione di una CNN

Per calcolare le attivazioni in una *CNN* si utilizza il prodotto di convoluzione al posto di un prodotto tra matrici:

$$\mathbf{Z} = \mathbf{W} * \mathbf{X} + \mathbf{b}$$

dove  $\mathbf{X}$  rappresenta l'immagine di ingresso con dimensioni corrispondenti ai pixel. La funzione di attivazione si trova allo stesso modo ( $\mathbf{A} = \Phi(\mathbf{Z})$ ) dei casi visti in precedenza. In generale, le immagini oggi sono codificate tramite codifica RGB (una immagine è composta da tre matrici, una per ciascuno dei tre colori principali, e il risultato finale è dato dalla somma di queste). Le *CNN*, che tipicamente lavorano con immagini a colori, effettuano la convoluzione su ognuna di queste matrici e sommano i risultati ottenuti (anche il *kernel* sarà composto da 3 matrici).

### 2.3.1 Dropout

Come accennato in precedenza, il dimensionamento di una *Rete Neurale* è un compito complesso, di solito si effettua in maniera empirica in quanto non è possibile determinare a priori la dimensione corretta. Una *Rete Neurale* troppo piccola potrebbe non essere in grado di riconoscere abbastanza tratti per classificare efficacemente gli ingressi, mentre una troppo grande potrebbe ottenere prestazioni estremamente buone in fase di addestramento ma ottenerne di pessime in fase di test, poiché durante l'addestramento ha sviluppato delle attivazioni nascoste che le permettevano di ottenere un punteggio più alto. È buona norma dimensionare una rete leggermente in eccesso (rispetto a quanto si pensa sia ottimale o a quanto risulta ottimale) per poi utilizzare il *Dropout*. Il *Dropout* è un metodo per evitare la creazione delle attivazioni nascoste: scegliendo una percentuale (di solito 50% ma può variare per applicazioni specifiche) questa diventerà la probabilità che un nodo si sconnetta da un altro. Il *Dropout* fa sì che la rete impari informazioni più robuste poiché, riducendo in maniera casuale il numero di connessioni, i nodi rimasti connessi dovranno regolare i propri pesi per adattarsi all'assenza dei nodi non connessi. È importante specificare che il *Dropout* ha effetto solo durante l'addestramento della *CNN* e non durante il test.

### 2.3.2 Loss functions

Le *Loss functions* sono funzioni utilizzate per decodificare l'output delle reti. Tre sono quelle più importanti:

- *Binary Crossentropy*, utilizzata per la classificazione binaria;
- *Categorical Crossentropy*, utilizzata per la classificazione multiclasse, restituisce un vettore con un 1 nella posizione del risultato più probabile;
- *Sparse Categorical Crossentropy*, utilizzata per la classificazione multiclasse, differisce dalla precedente per la rappresentazione: questa infatti restituisce un numero corrispondente alla classe più probabile.

### 2.3.3 Attivazioni

Esistono diversi tipi di attivazioni, nella Sezione 1.1.1 è stato visto l'output binario mentre, nella Sezione 1.1.3, è stato visto il sigmoide. Esistono altre attivazioni (ad esempio la funzione tangente iperbolica che viene spesso usata

---

Loss function	Usage	Examples	
		Using probabilities	Using logits
		<i>from_logits=False</i>	<i>from_logits=True</i>
BinaryCrossentropy	Binary classification	y_true: 1 y_pred: 0.69	y_true: 1 y_pred: 0.8
CategoricalCrossentropy	Multiclass classification	y_true: 0 0 1 y_pred: 0.30 0.15 0.55	y_true: 0 0 1 y_pred: 1.5 0.8 2.1
Sparse CategoricalCrossentropy	Multiclass classification	y_true: 2 y_pred: 0.30 0.15 0.55	y_true: 2 y_pred: 1.5 0.8 2.1

Figura 2.5: Loss Functions [1].

nelle *Reti Neurali* non convoluzionali) oltre a queste, quella che è stata utilizzata nelle prove sperimentali si chiama ReLU (Rectified Linear Unit). ReLU è una funzione che restituisce 0 ad ingresso negativo e il valore dell'ingresso in caso questi sia positivo:

$$\Phi(z) = \begin{cases} 0 & z < 0 \\ z & z \geq 0 \end{cases}$$

Questa sua struttura permette di avere una derivata sempre pari a 1, riportata all'ingresso, in caso di ingresso positivo.

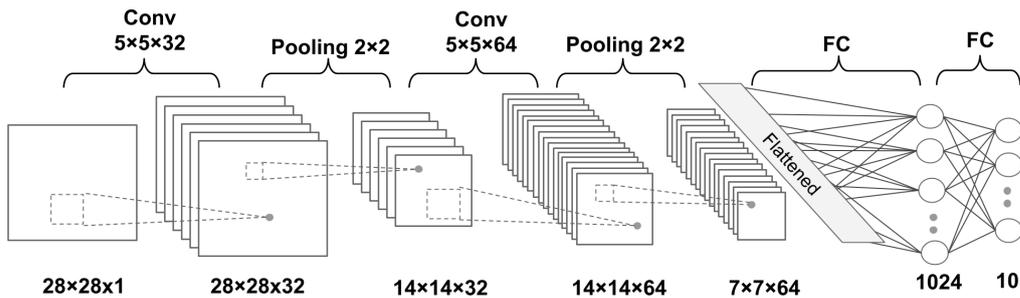


Figura 2.6: Esempio di CNN [1].

## Capitolo 3

# Python, Tensorflow e Keras

Normalmente, sono tre i principali linguaggi di programmazione utilizzabili per implementare algoritmi di *Machine Learning*: *Python*, *C++*, *Matlab*. Tutti questi linguaggi permettono, attraverso l'uso di apposite librerie o plugin, di effettuare i calcoli di interesse per le *CNN* (convoluzione 2D, derivate ecc).

### 3.1 Python

È stato scelto *Python* come linguaggio di programmazione per un motivo: la semplicità. *Python* è un linguaggio ad oggetti con la possibilità di creare classi, esattamente come in *C++*, tuttavia è nato con un'impostazione user-friendly che lo rende molto didattico. *Python* non raggiunge le prestazioni di *C++*, in quanto questi, grazie ai puntatori, può essere ottimizzato meglio lavorando a più basso livello. È pur vero che, grazie a librerie ottimizzate (scritte in *C++* anche per *Python*), la differenza di prestazioni oscilla tra il 5 e il 10%. *Matlab*, purtroppo, non possiede le librerie degli altri due linguaggi, rendendo perciò tutto il procedimento decisamente più complesso. La libreria più interessante, appartenente sia a *Python* che a *C++*, si chiama *Tensorflow*, una libreria che permette di rendere parallelo il calcolo tramite le GPU, e possiede una API chiamata *Keras* che verrà utilizzata per la creazione di un modello.

### 3.2 Tensorflow e Keras

*Tensorflow* è una libreria che permette di velocizzare significativamente i calcoli necessari ad un algoritmo di *Machine Learning*, contiene una serie di funzioni che sono in grado di sfruttare i *Tensori* [11]. *Keras* è una API di

*Tensorflow* molto utilizzata in *Machine Learning* per creare modelli di *CNN* con poche righe di codice. *Keras* possiede, inoltre, le funzioni necessarie a creare e salvare modelli di *CNN* per poter essere utilizzati successivamente in altre applicazioni (file del tipo modello.h5).

### 3.2.1 Utilizzo delle GPU

Le architetture delle CPU sono state oggetto di notevoli miglioramenti negli ultimi anni che hanno portato ad un incremento delle loro prestazioni. Il risultato è che le CPU sono in grado di eseguire codici complessi, anche se con dei limiti. Le architetture di calcolo parallelo delle GPU nVidia prendono il nome di architetture *CUDA* (Compute Unified Device Architecture) e posseggono i “CUDA cores”, simili ai core delle CPU. Nel caso venissero utilizzati algoritmi molto complessi e *Dataset* molto grandi, le CPU non sarebbero più in grado di addestrare il modello, con il risultato del programma terminato prima ancora di partire. A parità di costo le GPU offrono prestazioni migliori: un esempio potrebbe essere (i prezzi riportati sono relativi a dicembre 2019) una generica CPU Intel®Core™i9-9960X, uno dei top di gamma al tempo, che possiede sì una frequenza di clock di 3.1GHz, ma il basso numero di core (16 con 32 threads), la bassa velocità di trasferimento dati (79.47 GB/s), i soli 1290 GFLOPS (calcoli in virgola mobile) e il prezzo alto di circa 1700\$ non riescono a reggere il confronto, ad esempio, con una nVidia GeForce®RTX™2080Ti la quale sì possiede una frequenza di clock inferiore (1.35GHz), ma con 4352 cores, 616GB/s di velocità di trasferimento, i 13400GFLOPS a circa 600\$ in meno, risulta la scelta migliore [1].

### 3.3 Ubuntu 20.04.1 LTS

Viste le difficoltà riscontrate nella compatibilità di *Tensorflow* e *Keras* su sistema operativo Windows, si è pensato di utilizzarne uno Linux: Ubuntu. Ubuntu è una delle distribuzioni più conosciute ed user-friendly di Linux, basata su Debian. La stabilità è stato un fattore determinante per la scelta della distribuzione, infatti Ubuntu (nella sua versione 20.04.1) vanta non solo prestazioni migliorate ma una stabilità davvero esemplare. Tutte le distribuzioni Linux hanno una versione di *Python* preinstallata e, quindi, una compatibilità nativa con il linguaggio. Risulta, inoltre, molto facile installare ed aggiornare le librerie e il linguaggio stesso, con poche istruzioni e una serie di download il sistema operativo è pronto per lavorare. Purtroppo, durante la creazione del codice, è stata utilizzata una macchina virtuale per utilizza-

---

---

re Linux, e ciò ha ridotto considerevolmente le prestazioni ottenibili tramite l'utilizzo del solo processore.

### 3.4 Labelbinarizer

Prima di passare al lavoro svolto è giusto spendere qualche parola per il *Labelbinarizer*. Questa funzione (che appartiene a *sklearn*, una libreria molto utile per le applicazioni di *Machine Learning*) permette di “tradurre” la rappresentazione *one hot* ottenuta dalla *Loss function* in termini a noi più chiari. Si è deciso di utilizzare questa funzione, e non semplicemente di scrivere una lista contenente i nomi delle classi, per mantenere più generico il codice in modo da poterlo utilizzare con qualsiasi *Dataset*, purché rispetti delle condizioni che verranno discusse nella Sezione 4.3.



# Capitolo 4

## Implementazione Software e Prove Sperimentali

In questo capitolo viene presentato il lavoro svolto, a partire dalla preparazione dell'ambiente di lavoro fino alle misurazioni effettuate con diversi algoritmi. L'obiettivo è di creare un algoritmo per il riconoscimento di veicoli; verranno infatti comparate le prestazioni di vari algoritmi su due differenti *Dataset*, uno custom e uno pre-costruito, con classi relative alla tipologia dei veicoli (ad esempio “car” piuttosto del modello specifico).

### 4.1 Preparazione dell'ambiente di lavoro

Come già accennato nel Capitolo 3 si è deciso di lavorare con il sistema operativo Ubuntu. Ubuntu è stato installato su macchina virtuale nella sua versione più recente al momento della scrittura. Nonostante l'abbondante quantitativo di RAM reso disponibile per l'utilizzo (circa 12GB) la CPU comunque risultava condivisa tra due sistemi operativi, perciò le prestazioni sono state decisamente inferiori rispetto a quelle potenzialmente ottenibili in assenza di macchina virtuale. Una volta preparata la macchina virtuale, installare *Python* risulta molto semplice. È necessario installare il *pip* (serve per poter scaricare le librerie da terminale, nel caso specifico è stato installato nella versione *pip3*). Una volta fatto questo, per ottenere Tensorflow e Keras è sufficiente utilizzare le funzioni

```
pip3 install *nome_libreria*
```

Scrivendo al posto di *\*nome\_libreria\** “Tensorflow” e poi “Keras” in automatico verranno scaricate anche tutte le librerie necessarie al loro utilizzo. Verranno inoltre corrette incompatibilità di versioni di librerie installate. È

comunque consigliabile ripetere questa operazione se si vuole avere assoluta certezza di avere una certa libreria installata (ad esempio sklearn). Il codice è stato scritto utilizzando l'editor *gedit*.

## 4.2 Algoritmo VGG16

Prima di iniziare a scrivere il codice di addestramento, è necessario stabilire quale algoritmo si vuole addestrare. Attualmente, l'algoritmo considerato migliore per la classificazione di immagini è chiamato *VGG* (Visual Geometry Group). L'algoritmo viene descritto in maniera approfondita nel documento originale, pertanto si richiama alla lettura dello stesso [12]. Verrà ora riportato un esempio di algoritmo VGG16, consigliato anche nel blog di Keras, considerato uno dei migliori per la classificazione di immagini di grandi dimensioni :

```
model = Sequential()

model.add(Conv2D(input_shape=(224,224,3),
  filters=64,kernel_size=(3,3),
  padding="same", activation="relu"))
model.add(Conv2D(filters=64,kernel_size=(3,3),
  padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3),
  padding="same", activation="relu"))
model.add(Conv2D(filters=128, kernel_size=(3,3),
  padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=256, kernel_size=(3,3),
  padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3),
  padding="same", activation="relu"))
model.add(Conv2D(filters=256, kernel_size=(3,3),
  padding="same", activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3),
  padding="same",activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3),
```

---

```
padding="same",activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3),
padding="same",activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3),
padding="same",activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3),
padding="same",activation="relu"))
model.add(Conv2D(filters=512, kernel_size=(3,3),
padding="same",activation="relu"))
model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Flatten())
model.add(Dense(units=4096,activation="relu"))
model.add(Dense(units=4096,activation="relu"))
model.add(Dense(units=2, activation="softmax"))
```

Questo codice è tratto da un articolo dove viene spiegato come costruire un algoritmo di classificazione binaria su *Dataset* di cani e gatti [13]. Come si può notare dalle prime righe, il codice si addestra con immagini di  $224 \times 224$  pixel in formato RGB: questo richiede una potenza di calcolo molto elevata. Purtroppo la CPU utilizzata non è risultata in grado di computare un codice di questa complessità, perciò sono stati utilizzati codici meno complessi (immagini più piccole e meno unità per strato).

### 4.3 Creazione Dataset

Il codice di addestramento è stato scritto per risultare il più generico possibile, come spiegato nella sezione 3.4, in modo da poter utilizzare un Dataset custom. Infatti, come si evince dalla prima parte del codice (che verrà riportato nella sezione apposita), viene utilizzato il *Labelbinarizer* per poter assegnare, ad ogni elemento, il nome della cartella in cui è contenuto come classe [15]. Per funzionare in maniera corretta il *Labelbinarizer* necessita di una rappresentazione di tipo one-hot, questo spiega perchè, in tutti i codici, è stata scelta come *Loss-Function* la “categorical-crossentropy”. Così facendo è possibile riutilizzare il codice per addestrare la *Rete Neurale* per qualsiasi tipo di Dataset, purché abbia il contenuto diviso in cartelle corrispondenti alle classi di nostro interesse. Il Dataset è stato creato utilizzando vari motori di ricerca e raccogliendo un buon quantitativo di immagini per ogni classe (si parla nello specifico di 458 immagini per la classe “bike”, 520 per la classe

---

“car” e 315 per la classe “truck”) cercando di differenziarle il più possibile l’una dall’altra. La ricerca con parole chiave tipo “automobile”, “moto” o “camion” non ha condotto a risultati soddisfacenti, perciò si è deciso di effettuare ricerche per casa costruttrice (Audi, BMW, Wolkswagen, ecc.). La numerosità degli elementi delle singole classi risulta così diversa a causa del maggior numero di modelli di automobili, rispetto a camion o moto, prodotti negli anni. Il basso numero di campioni (come si vedrà nella sezione 4.3.2 i Dataset, di norma, hanno molto più di 1500 campioni) ha fatto sì che fosse necessario utilizzare l’istruzione

```
more = ImageDataGenerator(rotation_range=20,  
    width_shift_range=0.2,  
    height_shift_range=0.2, zoom_range=0.3,  
    horizontal_flip=True,  
    shear_range=0.2, fill_mode="nearest")
```

per poter aumentare il numero di campioni in maniera artificiosa. La classe *ImageDataGenerator* appartenente a *Keras* (per il funzionamento completo si rimanda al sito di Keras stesso) permette, tramite rotazioni, tagli, zoom, ecc., di modificare le immagini presenti nel Dataset e utilizzarne anche le versioni modificate per incrementare il numero di test effettuati: non risulta molto efficace con Dataset già grandi ma aiuta molto con i più piccoli come quello preso in esame.

### 4.3.1 Train Test Split

La funzione “train\_test\_split” è stata utilizzata per dividere il *Dataset*.

```
(train_X, test_X, train_Y, test_Y) = train_test_split(data,  
    labels, test_size=0.2)
```

Questa funzione infatti prende come ingressi i vettori, data e labels, e un “test\_size”: quest’ultimo parametro rappresenta una percentuale (espressa come numero compreso tra 0 e 1) che indica la quantità di dati del *Dataset* che verrà utilizzata per la fase di test, per ricavare quindi la parte che verrà utilizzata durante l’addestramento addestramento sarà sufficiente sottrarre il valore test\_size da 1 ( $1 - \text{test\_size}$ ). Da notare come la funzione ritorni quattro elementi, e non due come ci si aspetterebbe. Durante la fase di addestramento viene elaborato il vettore train\_X, per verificare se un risultato è o meno corretto, si compara il dato in uscita dalla rete con quello di train\_Y: se sono uguali allora il risultato è corretto, altrimenti il risultato necessita di correzioni. Stessa cosa avviene nella fase di test.

---

### 4.3.2 Utilizzo di un Dataset Pre-Costruito

Visti gli scarsi risultati ottenuti durante le prove effettuate con il Dataset custom, si è pensato di fare delle prove con un database già costruito. È stato trovato un Dataset, chiamato *MIO-TCD*, utilizzato come benchmark per la classificazione e la localizzazione di veicoli tramite *Computer Vision* e *Machine Learning* [14]. Questo Dataset è formato da una cartella contenente diverse sottocartelle (una per ogni classe) ciascuna chiamata con il nome della classe degli elementi che contiene. Esistono due versioni di questo Dataset, una per effettuare i benchmark sulla classificazione (648959 immagini divise in 11 classi) e una per quelli sulla localizzazione (137743 immagini divise in 11 classi). Le immagini sono state catturate dalle telecamere per controllare il traffico, presenti in tutto il territorio di USA e Canada, in diversi momenti della giornata e in diversi periodi dell'anno. Purtroppo l'enorme mole di dati è risultata non gestibile dall'architettura a disposizione. È stato possibile, tuttavia, ridurre la mole di dati all'interno del Dataset (riducendo il numero di classi e il numero di elementi all'interno di ognuna di esse) permettendo così anche ad una CPU poco performante, come quella considerata, di effettuare alcune prove.

## 4.4 Caricare le immagini

Per caricare le immagini è stato necessario utilizzare la libreria *Open CV*: le immagini vengono infatti caricate e ridimensionate per essere concordi con la dimensione della *CNN*. Questo frammento di codice

```
data = []
labels = []

imagePaths = sorted(list(paths.list_images('dataset')))
random.shuffle(imagePaths)
for imagePath in imagePaths:
    image = cv2.imread(imagePath)
    image = cv2.resize(image, (32, 32))
    image = img_to_array(image)
    data.append(image)

label = imagePath.split(os.path.sep)[-2]
labels.append(label)
```

---

```
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
```

```
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
```

permette di caricare le immagini dalle cartelle e creare, contemporaneamente, due array (data e labels) che contengono appunto le immagini e le classi a loro assegnate [16].

## 4.5 Grafici

Il codice, una volta finito di addestrare la *Rete Neurale Convolutionale*, utilizza alcuni dati raccolti per creare due grafici:

- accuratezza
- “loss ”

Entrambi questi grafici riportano due dati al loro interno, rispettivamente accuratezza durante l’addestramento e durante le prove, e “loss” durante l’addestramento e durante le prove. Con “loss” si intende la correzione: di quanto è stato necessario modificare l’uscita perchè risultasse corretta; in pratica rappresenta il *feedback* della rete per correggere i pesi. Viene riportato il codice di seguito:

```
x_arr=np.arange(len(H.history['loss']))+1

fig=plt.figure(figsize=(12,4))
ax1=fig.add_subplot(1,2,1)
ax1.plot(x_arr,H.history['accuracy'],'-o',label='Train accuracy')
ax1.plot(x_arr,H.history['val_accuracy'],'--<',
        label='Validation accuracy')
ax1.set_ylabel('Accuracy')
ax1.set_xlabel('Epochs')
ax1.legend(fontsize=15)

ax2=fig.add_subplot(1,2,2)
ax2.plot(x_arr,H.history['loss'],'-o',label='Train loss')
ax2.plot(x_arr,H.history['val_loss'],'--<',label='Validation loss')
ax2.set_ylabel('Loss')
```

---

```
ax2.set_xlabel('Epochs')
ax2.legend(fontsize=15)
plt.savefig('val_loss.jpg')
```

Questo codice, nello specifico, riporta l'andamento dei parametri precedentemente spiegati all'avanzare delle epoche.

## 4.6 Test Svolti

Come accennato in precedenza, sono stati svolti più test di algoritmi differenti con uno stesso Dataset. Al fine di mantenere l'elaborato conciso, verranno riportati i risultati dei test e l'algoritmo di *Machine Learning* senza però riportare il resto del codice.

### 4.6.1 Algoritmo didattico

Il primo algoritmo è estremamente semplice, viene riportato in uno dei libri utilizzati come esempio didattico [1]. Ha pochi strati (di fatto cinque in totale: 2 convolutivi, un *flatten* e due dense), le dimensioni del kernel sono standard e la dimensione dell'ingresso è contenuta rispetto agli altri codici (da  $28 \times 28$  a  $32 \times 32$  non sembra molto ma sono parecchi calcoli extra). La sua semplicità lo ha portato ad essere il codice più veloce nell'esecuzione (appena 20 secondi ad epoca).

```
model = Sequential()

model.add(Conv2D(32, (5,5), strides=(1,1),
    input_shape=(28,28,3),padding='same',
    data_format='channel_last'))
model.add(Activation("relu"))
model.add(MaxPooling2D((2,2)))

model.add(Conv2D(64, (5,5), strides=(1,1), padding='same' ))
model.add(Activation("relu"))
model.add(MaxPooling2D((2,2)))

model.add(Flatten())

model.add(Dense(1024))
model.add(Activation("relu"))
model.add(Dropout(0.5))
```

---

```
model.add(Dense(len(lbl.classes_)))
model.add(Activation("relu"))
```

Si era inizialmente pensato di riportare tre grafici per ogni codice, tuttavia risulterebbe ridondante perciò si è scelto di inserire nella trattazione solo quello da 100 epoche (Figura 4.1). Come si evince dalla Figura 4.1, l'algoritmo ha

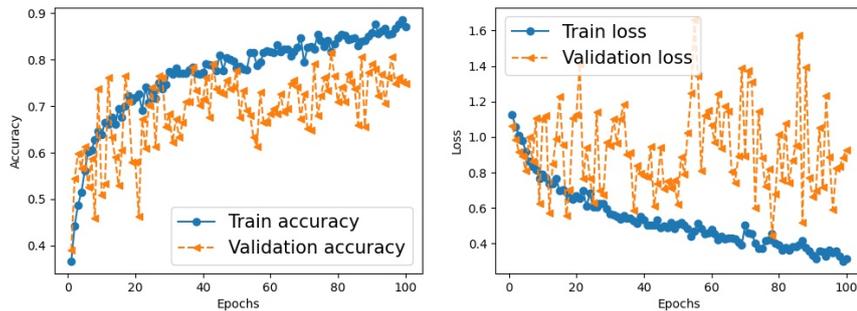


Figura 4.1: Prestazioni dell'algoritmo tratto dal libro.

restituito l'andamento atteso (Sezione 1.1) ossia logaritmico per la accuratezza. La funzione loss invece riporta un andamento esponenziale inverso. Importante notare come l'accuratezza cambi parecchio dall'addestramento al test. Questo è dovuto ad eventuali pattern contenuti nel *Dataset* (ad esempio la stessa immagine ripetuta o determinate immagini che sono in una sequenza semplificativa) in fase di addestramento che però, grazie anche al numero nettamente inferiore (20%!) di campioni, non sono presenti. Per quanto riguarda l'andamento altalenante della accuratezza nei test (validation accuracy) era stato, inizialmente, attribuito alla costruzione errata del *Dataset*. Il codice comunque riesce a raggiungere ottime prestazioni: infatti si notano picchi di accuratezza fino a 80%.

## 4.6.2 Algoritmi Custom

### Algoritmo custom 1

Questa è una prima versione di un algoritmo custom. Come si evince dalla Figura 4.2 l'algoritmo non è molto efficiente: con una accuratezza iniziale in fase di test molto bassa e costante e una accuratezza dei test finale decisamente più bassa di quella durante l'addestramento l'algoritmo risulta poco adatto a classificare immagini. È importante, tuttavia, notare come le immagini elaborate hanno dimensione  $32 \times 32 \times 3$ , perciò sono molto piccole.

La dimensione ridotta delle immagini e la complessità delle stesse (sono veicoli non, ad esempio, i numeri scritti a mano che spesso vengono usati in didattica) potrebbe giustificare i pessimi risultati ottenuti.

```
model.add(Conv2D(32, (5, 5),input_shape=(32,32,3),
padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D((3, 3)))
model.add(Dropout(0.5))

model.add(Conv2D(64, (5, 5),padding='same'))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(Conv2D(64, (5, 5),padding='same'))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D((3, 3)))
model.add(Dropout(0.5))

model.add(Conv2D(128, (4, 4),padding='same'))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(Conv2D(128, (4, 4),padding='same'))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(1024))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(len(lb.classes_)))
model.add(Activation("softmax"))
```

---

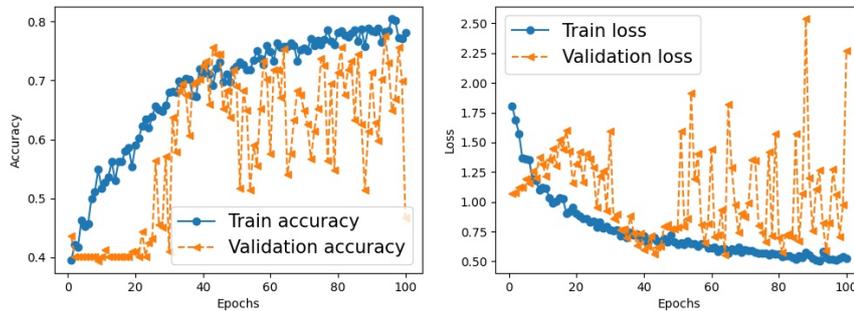


Figura 4.2: Prestazioni del primo algoritmo custom.

## Algoritmo custom 2

In questa seconda versione dell'algoritmo custom (Figura 4.3) si hanno comunque pessimi risultati. Tuttavia l'andamento sia di accuratezza che di loss risulta meno scostante, per quanto più rumoroso. Un buon punto a favore di questa versione è quello di avere una loss per lo più molto piccola con dei picchi nelle fasi finali dell'addestramento. Questo mostra come non sia necessario addestrare questo algoritmo oltre le 50-60 epoche: per quanto l'accuratezza in fase di addestramento aumenti, quella in fase di test rimane in un certo intorno di valori. Ancora una volta, risulta necessario ricordare come la dimensione ridotta per delle immagini complesse possa aver contribuito al mal funzionamento dell'algoritmo.

```

model.add(Conv2D(32, (4, 4), input_shape=(32,32,3),
padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D((3, 3)))
model.add(Dropout(0.5))

model.add(Conv2D(64, (4, 4), padding='same'))
model.add(Activation("relu"))
model.add(BatchNormalization())

model.add(Conv2D(64, (4, 4), padding='same'))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D((3, 3)))
model.add(Dropout(0.5))

```

---

```

model.add(Conv2D(128, (3, 3),padding='same'))
model.add(Activation("relu"))
model.add(BatchNormalization())

model.add(Conv2D(128, (3, 3),padding='same'))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())

model.add(Dense(1024))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(len(lb.classes_)))
model.add(Activation("softmax"))

```

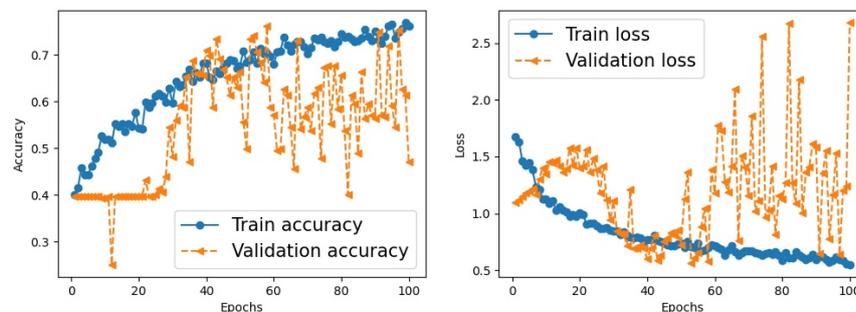


Figura 4.3: Prestazioni del secondo algoritmo custom.

### 4.6.3 Veichle Detection ConvNet [17]

Con questo algoritmo, tratto da [17], si ottengono risultati decisamente interessanti: il tempo di esecuzione è il più basso tra gli algoritmi testati e i valori di accuratezza nei test sono i più alti ottenuti. Per quanto riguarda il tempo di esecuzione (per gli algoritmi custom compreso tra i 45 e i 50 secondi a epoca) si parla di 20/21 secondi per epoca, paragonabile a quello didattico

tratto dal libro: un risultato ottimo considerando la complessità aggiuntiva rispetto a quello nella Sezione 4.6.1. Per quanto riguarda il valore di accuratezza nei test risulta poco rumoroso rispetto agli altri algoritmi testati e riesce ad ottenere dei picchi paragonabili a quelli in fase di addestramento. Addestrare questo algoritmo oltre le 100 epoche potrebbe portare dei benefici. Piccolo appunto per la funzione di loss, in questo caso è molto altalenante con valori anche estremamente alti, questo implica poca sicurezza durante la fase di addestramento; questo rappresenta tuttavia un aspetto marginale considerando gli ottimi risultati di accuratezza nei test. Come per gli altri algoritmi, un *Dataset* migliore e una dimensione delle immagini maggiore potrebbe portare benefici. Per il codice si rimanda a [17].

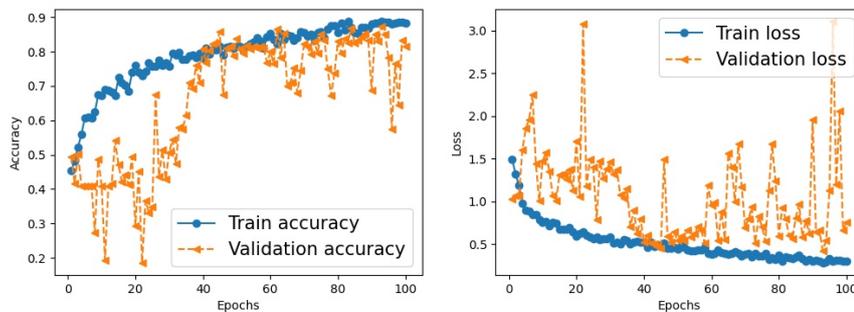


Figura 4.4: Prestazioni dell'algoritmo Veichle Detection ConvNet.

#### 4.6.4 Differenze di Prestazioni tra Dataset Custom e Pre-Costruito

Purtroppo, a causa delle limitate prestazioni dovute alla macchina virtuale, l'unico algoritmo che è stato possibile testare con il *Dataset TCD* ridotto (circa 45000 elementi) è quello tratto dal libro. I tempi di elaborazione, anche con un algoritmo a bassa complessità aumentano significativamente (si parla di circa 300 secondi per epoca). Per poter effettuare queste prove sono state necessarie alcune semplificazioni:

- Il *Dataset*, come già precedentemente accennato, è stato ridotto a 45000 elementi dai 600000 iniziali;
- Le classi, pur essendo meno delle 11 presenti nel *Dataset*, sono 4, poiché si hanno due tipi di “truck” differenti;
- Non è stata utilizzata la funzione “imageDataGenerator”;

- La batch size, che con l'altro *Dataset* era circa 50/70, con questo è stata settata a 7000 per ottenere dei tempi accettabili.

Fatta questa piccola premessa è ora possibile andare a visualizzare la Fig. 4.5, la quale mostra chiaramente un andamento poco rumoroso ed estremamente vicino tra addestramento e test. Il significativo aumento di prestazioni e la riduzione del rumore derivano da due fattori principali: l'aumento del numero di immagini nel *Dataset*, e la rimozione di `imageDataGenerator`. Con 45000 immagini divise tra addestramento e test, il numero di campioni a disposizione diventa significativamente più grande rispetto al *Dataset* custom che ne ha meno di 2000. La rimozione di `imageDataGenerator`, come si vedrà più nel dettaglio nella Sezione 4.6.5, riduce le oscillazioni di accuratezza e di loss nei test.

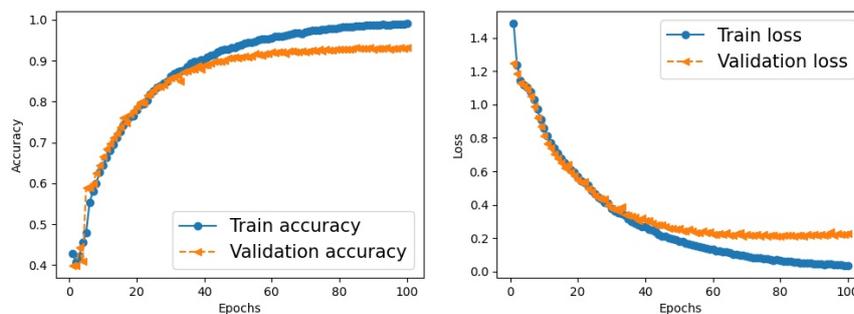


Figura 4.5: Prestazioni dell'algoritmo didattico su Dataset TCD.

#### 4.6.5 Rimozione di `ImageDataGenerator`

Uno dei primi dubbi sorti con i risultati ottenuti nella Sezione 4.6.4 è quello che la colpa possa essere dell'`ImageDataGenerator` poiché in [17] i risultati vengono più consoni alla Fig. 4.5 che non a quelle dei paragrafi precedenti e l'assenza di quella istruzione è una delle principali differenze. Si è allora deciso di effettuare delle prove sul *Dataset* custom per controllare i risultati ottenuti. Come si evince chiaramente dalla Fig. 4.6, non solo le prestazioni sono migliorate parecchio, non solo si è ottenuta una stabilità maggiore nei risultati, ma il picco massimo di accuratezza nei test (che comunque rimane sotto al 90%) è stato ottenuto circa nella epoca numero 50. Questa stabilità si può notare anche nel grafico della loss: nonostante questa aumenti con l'aumentare delle epoche (probabilmente dovuto al numero molto contenuto di elementi nel *Dataset*), comunque i risultati di accuratezza risultano più che accettabili. Dopo l'ottenimento di questi risultati, si è deciso di riprovare

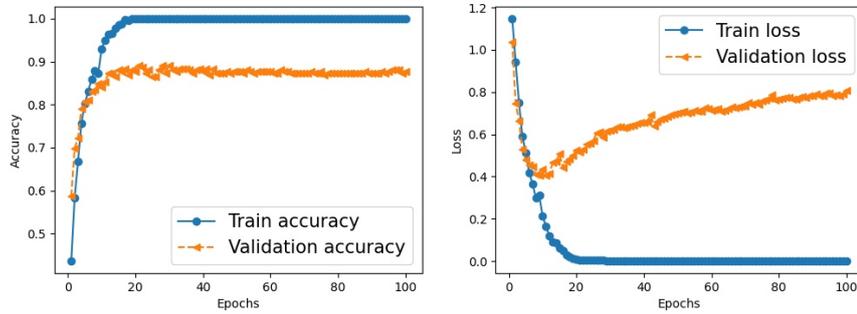


Figura 4.6: Prestazioni dell’algoritmo didattico su Dataset custom senza ImageDataGenerator.

anche l’algoritmo tratto da [17]. Anche in questo caso si può notare un chiaro

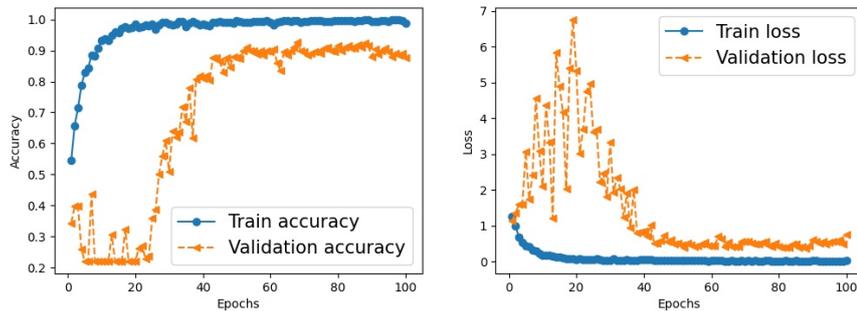
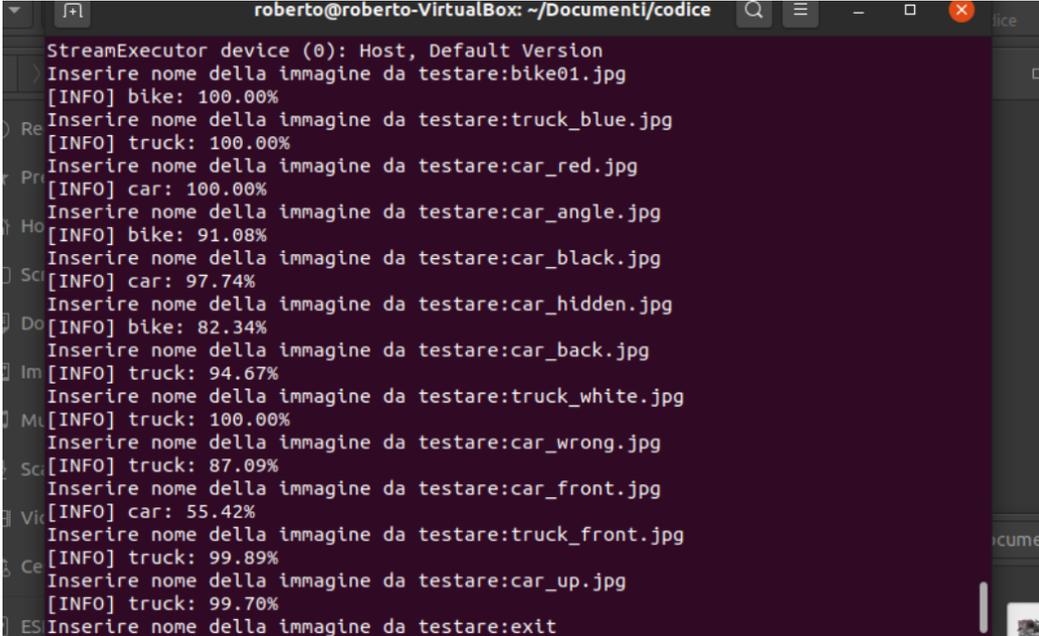


Figura 4.7: Performance di Veichle Detection ConvNet su Dataset custom senza ImageDataGenerator.

miglioramento rispetto ai risultati ottenuti nel paragrafo 4.6.3. Nonostante qui si abbiano comunque dei valori più altalenanti, soprattutto nelle prime fasi, questi tendono poi a stabilizzarsi su valori inferiori al 90% ma superiori al 80% nelle epoche dalla 50 in avanti. È molto interessante notare come, in fase di addestramento, l’accuratezza raggiunga il valore di 1: è inutile ripetere che sia un valore in realtà ottenibile soltanto in teoria e che le applicazioni reali, con le tecnologie odierne, non sono in grado di ottenerlo. Per concludere, i risultati suggeriscono che l’istruzione ImageDataGenerator provochi confusione all’algoritmo, e quindi risultati altalenanti.

### 4.6.6 Test con immagini

È stato utilizzato un breve codice per effettuare test sui vari modelli addestrati con i diversi algoritmi. Sono state riportate due immagini, quella



```
StreamExecutor device (0): Host, Default Version
Inserire nome della immagine da testare:bike01.jpg
[INFO] bike: 100.00%
Inserire nome della immagine da testare:truck_blue.jpg
[INFO] truck: 100.00%
Inserire nome della immagine da testare:car_red.jpg
[INFO] car: 100.00%
Inserire nome della immagine da testare:car_angle.jpg
[INFO] bike: 91.08%
Inserire nome della immagine da testare:car_black.jpg
[INFO] car: 97.74%
Inserire nome della immagine da testare:car_hidden.jpg
[INFO] bike: 82.34%
Inserire nome della immagine da testare:car_back.jpg
[INFO] truck: 94.67%
Inserire nome della immagine da testare:truck_white.jpg
[INFO] truck: 100.00%
Inserire nome della immagine da testare:car_wrong.jpg
[INFO] truck: 87.09%
Inserire nome della immagine da testare:car_front.jpg
[INFO] car: 55.42%
Inserire nome della immagine da testare:truck_front.jpg
[INFO] truck: 99.89%
Inserire nome della immagine da testare:car_up.jpg
[INFO] truck: 99.70%
ESInserire nome della immagine da testare:exit
```

Figura 4.8: Test con immagini dell'algoritmo didattico.

del terminale Linux e quella della cartella contenente le immagini testate, perchè, nonostante il codice sia progettato per mostrare a schermo le immagini con una label con scritta la classe e la percentuale di sicurezza, questa non viene mostrata. O meglio, non viene più mostrata nella data delle prove con i *Dataset* senza ImageDataGenerator, probabilmente a causa di qualche impostazione errata. In ogni caso la Fig. 4.8, in combinazione con la Fig. 4.9, mostra dati estremamente interessanti:

- La percentuale di sicurezza è del 100% per immagini tratte dal *Dataset* di addestramento (bike01.jpg) o per immagini simili a queste (car\_red.jpg);
- Con immagini nitide e scattate in maniera congrua a quelle del *Dataset* la sicurezza diminuisce ma rimane corretta e su valori alti (es. truck\_white.jpg, car\_black.jpg);
- Per immagini scattate male (es. car\_wrong.jpg) o molto diverse da quelle presenti nel *Dataset* (es. car\_hidden.jpg) i risultati sono sbagliati con una percentuale di sicurezza molto elevata.

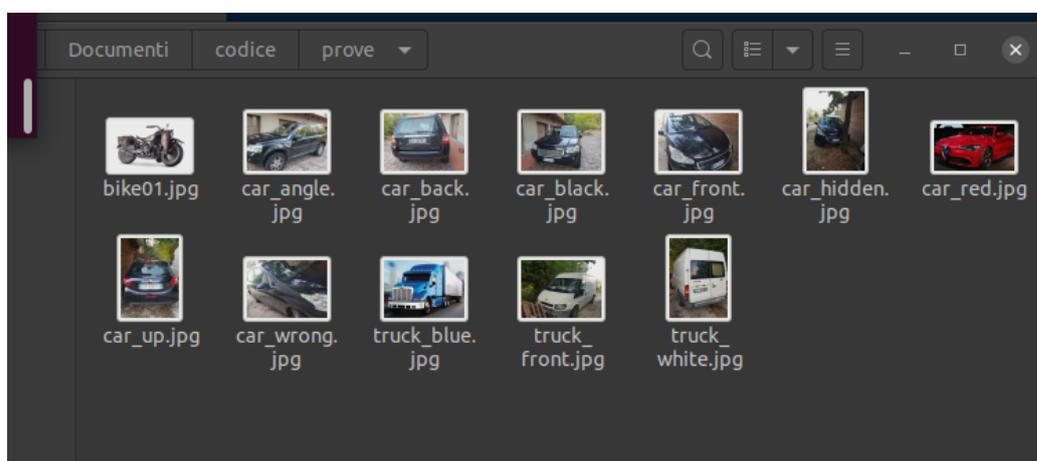


Figura 4.9: Immagini utilizzate nei test.

# Conclusioni

Questo elaborato tratta principalmente dei confronti tra vari algoritmi di *Reti Neurali Convolutionali* applicati ai medesimi *Dataset*, uno custom e uno pre-costruito.

Alla luce dei risultati ottenuti, illustrati nei grafici riportati nel Capitolo 4, si è concluso che l'istruzione *ImageDataGenerator* tenda a peggiorare le prestazioni della fase di addestramento degli algoritmi di *Machine Learning*, anche nel caso di *Dataset* di dimensioni ridotte. Infatti si nota chiaramente come l'accuratezza sia in fase di addestramento che in fase di test sia aumentata notevolmente dopo la rimozione di quella istruzione.

Dalle prove svolte si nota anche come, per algoritmi di questo tipo, il valore massimo di accuratezza nei test si ottenga nell'intorno dell'epoca 50. Per tale ragione continuare ad addestrare un codice oltre la 60/70<sup>a</sup> epoca non è efficace poiché i progressi sono quasi impercettibili. È importante specificare che per algoritmi più complessi, *Dataset* più grandi e soprattutto utilizzando una GPU è invece consigliabile farlo, anche solo per controllare a che punto questi raggiunge il valore massimo, un esempio di questo è nella Fig.4.5.

L'utilizzo di Python è consigliabile in virtù della quantità di informazioni e della documentazione disponibili in rete. È inoltre ormai considerato uno standard per il *Machine Learning* "homemade".

In combinazione con il linguaggio Python, è anche consigliabile l'uso di sistema operativo Linux. Questa scelta ha, infatti, permesso di risparmiare molto tempo nel setup dell'ambiente di lavoro e, inoltre, il grande quantitativo di informazioni di aiuto per questo tipo di sistemi operativi disponibile online rende il tutto di ancora più semplice utilizzo.

La scelta tra *Dataset* custom e pre-costruito dipende dalle condizioni di lavoro: se si conoscono le condizioni esatte di lavoro, e si ha la possibilità di

ottenere grandi moli di dati nelle stesse condizioni, allora risulta decisamente più efficiente.

Utilizzare una macchina virtuale per addestrare gli algoritmi si è rivelata una scelta tutt'altro che ottimale: CPU e RAM sono state divise tra i due sistemi operativi (con precedenza a quello principale) perciò i tempi di esecuzione sono aumentati parecchio e si hanno avuto grandi limiti alla quantità di dati processabili. Nello specifico, utilizzare un laptop potrebbe portare all'usura della batteria (spesso si hanno prestazioni ridotte quando non sono in ricarica) e ad uno sforzo eccessivo della CPU.

In generale, considerando il costo decisamente abbordabile di GPU molto potenti, si parla di 719,00 € per la scheda video nVidia GeForce® RTX™ 3080 (8704 CUDA cores), conviene utilizzare queste per addestrare gli algoritmi. Grazie al numero elevato di CUDA cores i tempi di calcolo si riducono di molto e le dimensioni dei *Dataset* possono essere aumentate senza preoccuparsi di sforzare la CPU (che può essere davvero poco performante a quel punto). È inoltre possibile utilizzare i CUDA cores anche in altre applicazioni (ad esempio MATLAB) perciò il costo viene compensato dall'utilizzo intensivo.

## Sviluppi futuri

Il *Machine Learning* rappresenta tutt'ora un argomento di ricerca molto caldo. Negli ultimi anni alcuni dei più grandi attori industriali nel campo dell'informatica hanno investito ingenti cifre nella ricerca e nello sviluppo di questa tecnologia. Nell'elaborato lo scopo era quello di classificare veicoli, in realtà il *Machine Learning* può essere applicato in contesti anche molto diversi, in base al tipo di dati per cui viene addestrato l'algoritmo. Un esempio è dato dalle applicazioni mediche dove sta diventando sempre più integrato l'utilizzo di questi algoritmi per riconoscere malattie o mutazioni. Il *Machine Learning* è, inoltre, la base per l'*Intelligenza Artificiale*, che viene utilizzata per facilitare l'interazione uomo macchina (assistente vocale Google, Siri, Cortana ecc.), e lo studio dello *Human Behaviour*. Le applicazioni sono molteplici. È importante aggiungere il *Quantum Computing* al quadro generale: infatti si pensa che su questi calcolatori sarà possibile addestrare algoritmi estremamente più complessi di quelli odierni, per aumentare la accuratezza. Al giorno d'oggi, purtroppo, il *Machine Learning* è limitato al tipo di dato in cui viene addestrato l'algoritmo, un buon miglioramento potrebbe essere riuscire ad addestrarne su più argomenti in modo da ridurre il numero di modelli da utilizzare.

---

# Codice

```
import matplotlib
matplotlib.use("Agg")
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras import backend as K
from keras.constraints import maxnorm
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import img_to_array
from sklearn.preprocessing import LabelBinarizer
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from imutils import paths
import numpy as np
import argparse
import random
import pickle
import cv2
import os

EPOCHS=int(input("Digitare numero di 'epoche' per l'addestramento:"))

BATCH_SIZE=70
```

```
data = []
labels = []

imagePaths = sorted(list(paths.list_images('dataset')))
random.shuffle(imagePaths)

for imagePath in imagePaths:
    image = cv2.imread(imagePath)
    image = cv2.resize(image, (32, 32))
    image = img_to_array(image)
    data.append(image)

    label = imagePath.split(os.path.sep)[-2]
    labels.append(label)

data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)

lb = LabelBinarizer()
labels = lb.fit_transform(labels)

(train_X, test_X, train_Y, test_Y) = train_test_split(data,
labels, test_size=0.2)

model = Sequential()

model.add(Conv2D(32, (5,5), strides=(1,1),
    input_shape=(28,28,3),padding='same',
    data_format='channel_last'))
model.add(Activation("relu"))
model.add(MaxPooling2D((2,2)))

model.add(Conv2D(64, (5,5), strides=(1,1), padding='same' ))
model.add(Activation("relu"))
model.add(MaxPooling2D((2,2)))

model.add(Flatten())
```

---

```
model.add(Dense(1024))
model.add(Activation("relu"))
model.add(Dropout(0.5))

model.add(Dense(len(lb.classes_)))
model.add(Activation("relu"))

adam = Adam(lr=1e-3,decay=((1e-3)/EPOCHS))
model.compile(loss="categorical_crossentropy",
              optimizer=adam, metrics=["accuracy"])

H=model.fit(train_X, train_Y, batch_size=BATCH_SIZE,
            validation_data=(test_X, test_Y),
            epochs=EPOCHS)

model.save("vehicle_model", save_format="h5")

f = open("vehicle_labelbin", "wb")
f.write(pickle.dumps(lb))
f.close()

x_arr=np.arange(len(H.history['loss']))+1

fig=plt.figure(figsize=(12,4))
ax1=fig.add_subplot(1,2,1)
ax1.plot(x_arr,H.history['accuracy'],'-o',label='Train accuracy')
ax1.plot(x_arr,H.history['val_accuracy'],'--<',
        label='Validation accuracy')
ax1.set_ylabel('Accuracy')
ax1.set_xlabel('Epochs')
ax1.legend(fontsize=15)

ax2=fig.add_subplot(1,2,2)
ax2.plot(x_arr,H.history['loss'],'-o',label='Train loss')
ax2.plot(x_arr,H.history['val_loss'],'--<',label='Validation loss')
ax2.set_ylabel('Loss')
```

---

```
ax2.set_xlabel('Epochs')  
ax2.legend(fontsize=15)  
plt.savefig('val_loss.jpg')
```

# Elenco delle figure

1.1	Output binario di $\theta(z)$ [1]. . . . .	5
1.2	Differenze tra <i>Perceptron</i> e <i>Adaline</i> [1]. . . . .	6
1.3	Sigmoid Function [1]. . . . .	7
1.4	Esempio di <i>Rete Neurale</i> con più “ <i>Hidden Layers</i> ” . . . . .	9
2.1	Esempio di zero padding con l’aggiunta di 4 “0” [1]. . . . .	15
2.2	Output con i 3 tipi di padding [1]. . . . .	16
2.3	Convoluzione 2D con same padding [1]. . . . .	17
2.4	Pooling [1]. . . . .	18
2.5	Loss Functions [1]. . . . .	20
2.6	Esempio di CNN [1]. . . . .	20
4.1	Prestazioni dell’algoritmo tratto dal libro. . . . .	32
4.2	Prestazioni del primo algoritmo custom. . . . .	34
4.3	Prestazioni del secondo algoritmo custom. . . . .	35
4.4	Prestazioni dell’algoritmo Veichle Detection ConvNet. . . . .	36
4.5	Prestazioni dell’algoritmo didattico su Dataset TCD. . . . .	37
4.6	Prestazioni dell’algoritmo didattico su Dataset custom senza ImageDataGenerator. . . . .	38
4.7	Performance di Veichle Detection ConvNet su Dataset custom senza ImageDataGenerator. . . . .	38
4.8	Test con immagini dell’algoritmo didattico. . . . .	39
4.9	Immagini utilizzate nei test. . . . .	40



# Bibliografia

- [1] Sebastian Raschka & Vahid Mirjalili: *Python Machine Learning third edition, Packt*,(2019)
- [2] W. S. McCulloch & W. Pitts: *A Logical Calculus of the Ideas Immanent in Nervous Activity, Bulletin of Mathematical Biophysics, 5(4): 115-133*, (1943)
- [3] Frank Rosenblatt: *The Perceptron: A Perceiving and Recognizing Automation, Cornell Aeronautical Laboratory*,(1957)
- [4] B. Widrow, T. Hoff, altri: *An Adaptive "Adaline" Neuron Using Chemical "Memristors", Technical Report Number 1553-2, Stanford, CA*,(1960)
- [5] Jason Brownlee: *Gentle Introduction To The Adam Optimization Algorithm For Deep Learning, Machine Learning Mastery*,(2017)
- [6] Diederik Kingma & Jimmy Ba: *Adam: A Method For Stochastic Optimization, University of Toronto, ICLR*,(2015)
- [7] Wikipedia: *AI Winter*
- [8] Wikipedia: *Recurrent Neural Network*
- [9] Y. LeCun e team: *Handwritten Digit Recognition With A Back-Propagation Network, NeurIPS conference*,(1989)
- [10] David H. Hubel: *Our First Paper, on Cat Cortex, Oxford website*,(1959)
- [11] Wikipedia: *Tensore*
- [12] Karen Simonyan, Andrew Zisserman: *Very Deep Convolutional Neural Network for Large-Scale Image Recognition, Cornell University*,(2014)
- [13] Rohit Tahkur: *Step by Step VGG16 Implementation for Beginners, towardsdatascience.com*,(2019)

- [14] Z. Luo, F.B. Charron, C. Lemaire, J. Konrad, S. Li, A. Mishra, A. Achkar, J. Eichel, P-M Jodoin: *MIO-TCD: A new benchmark dataset for vehicle classification and localization*, in press at *IEEE Transaction on Image Processing*,(2018)
- [15] Adrian Rosenbrock: *Keras and Convolutional Neural Networks (CNN)*, *pyimagesearch*,(2018)
- [16] “AstarLight”: *xiao-xiao chen, Keras image classifier framework, GitHub*,(2017)
- [17] Benedetta Baldini: *Implementazione in Python e Tensorflow di una Rete Neurale Convolutionale per la Classificazione di Veicoli, Tesi UniBo*,(2019)