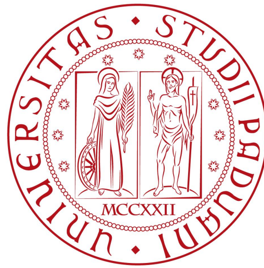
Università degli Studi di Padova
Dipartimento di Scienze Statistiche
Corso di Laurea Magistrale in Scienze Statistiche



CONVOLUTIONAL NEURAL NETWORKS PER IL RICONOSCIMENTO DI NUDITA' NELLE IMMAGINI

Relatore: prof. Bruno Scarpa
Dipartimento di Scienze Statistiche

Laureando: Elia Giacomini
Matricola n. 1111293

Anno Accademico 2016/2017

Indice

1 Introduzione **11**

2 Metodi Classici **13**

2.1 Strumenti statistici 13

2.1.1 Bagging 13

2.1.2 Foreste Casuali 14

2.1.3 Gradient Boosting 15

2.2 Estrazione delle *features* discriminanti 17

2.2.1 Riconoscimento della pelle 18

2.2.2 Localizzazione dei volti 19

3 Reti Neurali **23**

3.1 Motivazione biologica e connessioni 23

3.2 Architettura di una rete neurale 24

3.3 Stima di una rete neurale 26

3.3.1 Calcolo del gradiente: l'algoritmo di *back-propagation* 27

3.3.2 Stochastic Gradient Descent (SGD) 28

3.3.3 Algoritmi di ottimizzazione della discesa del gradiente 29

3.3.4 Forme di regolarizzazione 34

3.4 Scelte di non linearità 37

3.4.1 Funzione Sigmoide 37

3.4.2 Tangente iperbolica 38

3.4.3 ReLU 39

3.4.4 Leaky ReLU 39

3.5 Inizializzazione dei pesi 40

3.5.1 Batch Normalization 41

4 Convolutional Neural Networks **47**

4.1 L'operazione di convoluzione 48

4.2 Strati di una CNN 50

4.2.1 Convolutional Layer 50

4.2.2 Pooling Layer 54

4.2.3 Fully-Connected Layer 54

4.3 Architettura generale della rete 55

4.4 Il modulo *Inception* . . . 56
4.5 Data Augmentation . . . 58

5 Dataset **59**
5.1 NPDI Database . . . 59
5.2 Web Crawling . . . 61

6 Analisi e Modelli **63**
6.1 Primi modelli sul dataset NPDI . . . 63
6.1.1 Gradient Boosting . . . 64
6.1.2 Convolutional Neural Networks . . . 67
6.2 Analisi sul "*Crawling* dataset" . . . 69
6.2.1 Data Augmentation . . . 70
6.2.2 Più strati, filtri più piccoli . . . 70
6.2.3 Architettura *Inception* . . . 71

7 Aspetti Computazionali **75**
7.1 Configurazione software . . . 75
7.2 Calcolo parallelo e sfruttamento della GPU . . . 75
7.3 *Iterator* e determinazione del *batch size* . . . 76

8 Conclusioni **79**

Bibliografia **81**

Elenco delle figure

2.1 Operazione di confronto su un singolo pixel. 20
2.2 Esempi di vicini circolari di diverso raggio (R) e numerosità (P). Il valore dei punti che cadono all'interno di più pixel viene interpolato. 21
2.3 Esempi di vicini circolari di diverso raggio (R) e numerosità (P). Il valore dei punti che cadono all'interno di più pixel viene interpolato. 21
2.4 L'output finale dell'operatore LBP coincide con l'istogramma ottenuto attraverso la concatenazione degli istogrammi prodotti nelle singole regioni. 21

3.1 Illustrazione biologica di un neurone (sinistra) e rappresentazione del relativo modello matematico (destra). Fonte: Karpathy (2015). . . . 24
3.2 Diagramma di una rete neurale con un singolo strato nascosto. . . . 24
3.3 Diagramma di una rete neurale con tre strati nascosti e output multipli (Efron e Hastie, 2016). 25
3.4 Fluttuazioni SGD con dimensione *batch* pari ad uno. 29
3.5 Discesa del gradiente senza (a) e con (b) ottimizzazione *momentum*. 30
3.6 Mentre il *momentum* calcola prima il gradiente corrente (freccia piccola blu) per poi saltare in direzione del gradiente accumulato (freccia grande blu), il NAG esegue prima un salto in direzione del gradiente accumulato in precedenza (freccia marrone), e successivamente calcola il gradiente corrente, applicando una correzione (freccia verde). 30
3.7 Performance degli algoritmi di ottimizzazione SGD sui livelli di una superficie di perdita. Fonte: Ruder (2016). 33
3.8 Performance degli algoritmi di ottimizzazione SGD su un punto di sella. 34
3.9 Contorni della regione imposta da $J(W) = \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{l=1}^{p_k-1} \left| w_{l_j}^{(k)} \right|^q$, per differenti valori di q. 35
3.10 Contorni della regione imposta da $J(W) = \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{l=1}^{p_k-1} \left| w_{l_j}^{(k)} \right|^q$ con $q = 2$ (sinistra), elastic-net con $\alpha = 1.2$ (destra). Nonostante i due contorni appaiano simili, solo l'elastic-net risulta non differenziabile negli angoli. 35

3.11 Diagramma di una rete neurale prima (sinistra) e dopo (destra) l'applicazione del *dropout*. I nodi segnati con una croce sono quelli che sono stati scelti casualmente per essere rimossi dal modello. 36
3.12 Sinistra: il neurone durante la fase di stima è presente con probabilità ϕ, ed è connesso con i neuroni dello strato successivo attraverso i pesi w. Destra: lo stesso neurone in fase di test è sempre presente nella rete e i suoi pesi sono moltiplicati per ϕ. 36
3.13 Esempio di dinamica a zig-zag nell'aggiornamento di due pesi W_1 e W_2. La freccia blu indica indica l'ipotetico vettore ottimale per la discesa del gradiente, mentre le frecce rosse i passi di aggiornamento compiuti: gradienti tutti dello stesso segno comportano due sole possibili direzioni di aggiornamento. 38

4.1 Operazione di convoluzione nel caso bidimensionale: un filtro di dimensione 2×2 viene moltiplicato elemento per elemento per una porzione dell'input di uguali dimensioni. L'output dell'operazione è costituito dalla somma di tali prodotti. L'operazione di convoluzione viene infine ripetuta spostando il filtro lungo le due dimensioni dell'input. 49
4.2 Operazione di convoluzione di un filtro di dimensioni 5×5 e relativa mappa di attivazioni prodotta. 50
4.3 Operazione di convoluzione di due differenti filtri ($W0$ e $W1$) di dimensione $3 \times 3 \times 3$ su un volume di input $7 \times 7 \times 3$. 51
4.4 Set di 96 filtri $11 \times 11 \times 3$ appresi dall'architettura di Krizhevsky *et al.* (2012) in un problema di classificazione di immagini. L'assunzione di *weight sharing* è ragionevole dal momento che individuare una linea o uno spigolo è importante in qualsiasi posizione dell'immagine, e di conseguenza non c'è la necessità di imparare ad localizzare la stessa caratteristica in tutte le possibili zone. 53
4.5 Esempio di max-pooling: il filtro opera indipendentemente su ogni *features-map* e di conseguenza la profondità del volume rimane inalterata. Larghezza e altezza vengono ridimensioante invece di un fattore $F = 2$ (di conseguenza viene eliminato il 75% dei pesi). 54
4.6 Esempio di apprendimento delle *features* su un dataset di oggetti misti. 55
4.7 Architettura del modulo *inception*. 56

5.1 Alcune immagini del database, divise per categoria in base alla riga. 60
5.2 Esempio di immagini che compongono il dataset PIPA. 62
5.3 Alcune delle 410 attività rappresentate nel database MPII Human Pose dataset, divise per categoria (colonne). 62

6.1 Esempio di localizzazione del volto. 66
6.2 Maschera binaria dei pixel identificati come pelle (colore bianco). . . 66

6.3 Curve di densità relative alla percentuale di pelle nelle immagini, divise in base alla categoria di appartenenza. 67
6.4 Errore di classificazione per training e test set, in funzione dell'iterazione. Il dataset utilizzato è quello intero, ovvero con composizione mista. 67
6.5 Correlogramma spaziale calcolato sulle immagini del dataset NPDI. . 68

Elenco delle tabelle

5.1 Statistiche dei video dai quali sono state campionate le immagini. . . 60
5.2 Etnia degli attori nei video pornografici 60

6.1 Diverse modalità valutate per il ridimensionamento delle immagini, a partire dalla versione originale (in alto). 64
6.2 Risultati ottenuti con un modello gradient boosting sui singoli pixel per diverse composizioni del dataset. 65
6.3 Risultati ottenuti con un modello gradient boosting sui singoli pixel con l'aggiunta delle variabili relative alle informazioni sulla pelle. . . . 66
6.4 Architettura disegnata sulla falsariga di quella proposta da Simonyan e Zisserman (2014) e ridimensionata all'hardware a disposizione. . . . 71
6.5 Percentuale di corretta classificazione delle reti sui due diversi test set. 72
6.6 Performance del modello finale pre-allenato sul dataset ILSVRC1000. 73

7.1 Prestazioni di diverse architetture CPU e GPU sulla rete *Inception-BN* 76
7.2 Calcolo del numero di parametri e di neuroni necessario per ogni strato della rete. I due numeri dopo l'underscore indicano rispettivamente estensione spaziale e numero dei filtri. 77

Capitolo 1

Introduzione

Non è un segreto che, dall'avvento di internet, i materiali a contenuto pornografico siano divenuti alla portata di click da parte di tutti, bambini e minori compresi. La situazione si è aggravata ulteriormente in seguito alla recente diffusione di smartphone e tablet sempre connessi in rete, e alla parallela crescita esponenziale di tali contenuti disponibili online. Isolare una mole così ampia di materiale inappropriato risulta però essere un lavoro improponibile da svolgere manualmente (basti pensare al flusso continuo di immagini e video caricati sulla rete attraverso i vari social network), ragion per cui si è resa necessaria l'implementazione di sistemi in grado di rilevare in modo automatico la presenza di contenuti espliciti in immagini e video, e filtrarli di conseguenza in base all'utente destinatario. Tale sfida, che si traduce in un problema di classificazione binaria, verrà affrontata in questo elaborato attraverso dapprima l'utilizzo degli strumenti classici offerti dalla statistica, per entrare successivamente nel campo del *deep learning*, in particolare con l'applicazione delle *Convolutional Neural Networks.* Questa scelta sarà dettata come vedremo non solo dai limiti imposti dai metodi tradizionali, ma anche dai problemi che questi implicano dal punto di vista computazionale quando si tratta di passare all'atto pratico, congiuntamente a quelli imposti dai software esistenti per affrontare questo tipo di analisi.

Nel primo capitolo verranno presentati i principali strumenti statistici che sono stati utilizzati, corredati da alcuni algoritmi per l'estrazione delle principali *features* discriminanti in problemi di riconoscimento di nudità, ovvero il riconoscimento della pelle e la localizzazione dei volti; questo con lo scopo sia di fornire delle misure di sintesi in grado di incrementare la capacità predittiva dei modelli statistici, sia di mostrare i limiti dei metodi storici che si basano su queste caratteristiche. É bene comunque precisare che questi modelli sono stati presentati principalmente per avere un termine di paragone, e che nel campo della *Computer Vision* esistono algoritmi e descrittori di *features* molto più sofisticati, che però esulano dai contenuti trattati in questo corso di studi.
Il secondo capitolo introduce le reti neurali soffermandosi in particolare sugli accorgimenti necessari per garantire un allenamento efficiente, mentre il terzo capitolo entra nel cuore del modello proposto, cercando di spiegare in dettaglio il funziona-

mento delle *Convolutional Neural Networks* e le motivazioni che risiedono alla base di questa architettura.
I capitoli successivi presentano lo svolgimento delle analisi e i risultati ottenuti sui diversi dataset che si sono utilizzati, mentre il capitolo conclusivo si sofferma sugli aspetti computazionali, i quali si sono rivelati di fondamentale importanza per la sostenibilità dal punto di vista pratico delle analisi.

Capitolo 2

Metodi Classici

In questo primo capitolo viene effettuata una rassegna tematica degli strumenti statistici tradizionali che sono stati utilizzati durante le prime analisi. La loro presenza o meno all'interno di questa sezione è legata per lo più a ragioni computazionali oltre che di adeguatezza alla risoluzione del problema proposto. Sono infatti sopravvissuti alla selezione solo quei modelli che prevedono tempi e costi computazionali accessibili in relazione all'hardware a disposizione. In altre parole, quei metodi che possono essere implementati sfruttando la parallelizzazione e/o stimati in maniera ricorsiva.

2.1 Strumenti statistici

2.1.1 Bagging

Il *Bagging*, o *Bootstrap aggregation*, è una procedura *ensemble* disegnata per migliorare la stabilità e la precisione di un modello statistico applicato in contesti di classificazione o di regressione. Dato un training set Z di n osservazioni, il *bagging* opera effettuando B campionamenti *bootstrap* di Z ed adattando ad ognuno il modello in questione. Si ottengono così B stime di previsione che, una volta mediate, andranno a costituire la stima *bagging*:

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x) \tag{2.1}$$

dove $\hat{f}^{*b}(x)$ è il modello adattato al campione *bootstrap* Z^{*b}, $b = 1, ..., B$. La stima *bagging* (2.1) sarà diversa dalla stima originale $\hat{f}(x)$ solo se quest'ultima è funzione non lineare dei dati: considerando un modello con componente lineare e componenti di ordine maggiore, il *bagging* lascia invariata la parte lineare ma riduce la variabilità delle altre componenti rimpiazzandole con un'approssimazione empirica dei loro valori attesi (per maggiori approfondimenti algebrici si veda Friedman e Hall (2000)). Questo implica che più è lineare lo stimatore e meno efficace risulterà l'applicazione del *bagging*. Dall'altro lato, modelli altamente non lineari come reti

neurali ed alberi ricevono un grande beneficio in termini di riduzione della varianza, mantenendo intatta la distorsione. In particolare, l'instabilità che caratterizza gli alberi li rende un soggetto ideale per questo tipo di procedura, dal momento che sono affetti da una grande quantità di varianza su cui il *bagging* può lavorare (piccoli cambiamenti nei dati possono condurre ad una serie di *split* totalmente differente e la loro struttura gerarchica comporta la propagazione di eventuali errori ai primi stadi).

2.1.2 Foreste Casuali

Le foreste casuali sono una modifica sostanziale del *bagging* in cui viene costruita una collezione di alberi incorrelati. L'idea alla base delle foreste casuali può essere presentata pensando di dover effettuare la media di B variabili casuali ognuna con varianza σ^2, identicamente distribuite ma non necessariamente indipendenti, in particolare con indice di correlazione ρ. La varianza della loro media sarà pari a

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \tag{2.2}$$

All'aumentare di B il secondo termine scompare, ma il primo rimane ed il suo valore risulta direttamente proporzionale all'indice di correlazione. L'intuizione delle foreste casuali è quindi quella di migliorare la riduzione della varianza introdotta dal *bagging* attraverso una diminuzione della correlazione degli alberi. A tale scopo, questi vengono fatti crescere attraverso una selezione casuale delle variabili esplicative; in particolare, per ogni albero e prima di ogni *split* viene selezionato un numero $m < p$ di variabili candidate per la successiva suddivisione (algoritmo 2.1). Tipicamente, per problemi di classificazione si consigliano valori di m pari a $\sqrt{p}$.

Algorithm 2.1 FORESTE CASUALI PER PROBLEMI DI CLASSIFICAZIONE

1. **for** $b = 1, ..., B$ **do:**
   (a) Ottenere un campione *bootstrap* Z^* dal training set.
   (b) Stimare un albero T_b sul campione appena ottenuto, ripetendo in maniera ricorsiva i seguenti *step*:
      i Selezionare casualmente m variabili dalle p a disposizione.
      ii Effettuare lo *split* che rende minima la funzione di perdita.
2. Restituire l'insieme di alberi $\{T_b\}_1^B$ ottenuti al punto 1.
3. Previsione su una nuova osservazione x: Sia $\hat{C}_b(x)$ la classe prevista dal *b-esimo* albero. La previsione finale coincide con il voto di maggioranza di tutte le $\left\{\hat{C}_b(x)\right\}_1^B$ previsioni.

2.1.3 Gradient Boosting

Il *boosting* è un altro algoritmo di apprendimento supervisionato di tipo *ensemble* fondato sull'idea di Valiant (1984) che si possa costruire un classificatore forte attraverso la combinazione di classificatori deboli, ossia di classificatori la cui accuratezza è leggermente migliore della scelta casuale. Tale combinazione avviene secondo una procedura iterativa che prevede di stimare in maniera sequenziale i classificatori deboli su versioni ripetutamente modificate dei dati, in maniera analoga a quanto effettuato da AdaBoost.M1 (Freund *et al.* (1997), per una descrizione dettagliata si veda l'algoritmo 2.2).

Algorithm 2.2 AdaBoost.M1

---

1. Inizializzare i pesi di ogni osservazione $w_i = 1/N, i = 1, ..., N$
2. **for** $m = 1, ..., M$ **do:**
   (a) Stimare un classificatore debole $G_m(x)$ ai dati pesati secondo w_i.
   (b) Calcolare
   $$err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$$
   (c) Calcolare $\alpha_m = log\left(\frac{(1-err_m)}{err_m}\right)$
   (d) Aggiornare i pesi aumentando l'influenza delle osservazioni classificate scorrettamente e viceversa:
   $$w_i \leftarrow w_i \cdot exp[\alpha_m \cdot I(y_i \neq G_m(x_i))], \qquad i = 1, ..., N$$
3. **Output:** $G(x) = sign\left[\sum_{m=1}^{M} \alpha_m G_m(x)\right]$

Più in generale, il *boosting* può essere visto come un modo di stimare una generica funzione definita attraverso una sommatoria di funzioni di base:

$$f(x) = \sum_{m=1}^{M} \beta_m b(x; \gamma_m) \tag{2.3}$$

dove la scelta di b può spaziare dallo strato nascosto di una rete neurale ($b(x; \gamma) = \sigma(\gamma_0(+\gamma_1^T x))$, capitolo 3), alla regola di crescita di un albero, con γ che parametrizza gli *split*.

Tipicamente, la stima dell'insieme di parametri (β_m, γ_m) avviene attraverso la minimizzazione di una determinata funzione di perdita L:

$$\begin{aligned} \underset{\{\beta_m,\gamma_m\}_1^M}{min} &= \sum_{i=1}^{N} L(y_i, f(x_i)) \\ &= \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b(x;\gamma_m)\right) \end{aligned} \tag{2.4}$$

Per la maggior parte delle funzioni di perdita, risolvere la 2.4 richiede però costi computazionali insostenibili, motivo per il quale si preferisce un approccio *Forward Stagewise* che consiste nell'approssimare la 2.4 attraverso minimizzazioni successive, dove ad ogni passo viene aggiunta una nuova funzione di base (algoritmo 2.3).

Algorithm 2.3 Forward Stagewise Additive Modeling

1. Inizializzare $f_0(x) = 0$
2. **for** $m = 1, ..., M$ **do:**
   (a) Calcolare
   $$(\beta_m, \gamma_m) = \underset{\beta,\gamma}{argmin} \sum_{i=1}^{N} L(y_i, f_{m-1}(x_i) + \beta b(x_i;\gamma))$$
   (b) Impostare $f_m(x) = f_{m-1}(x) + \beta_m b(x;\gamma_m)$

In particolare, si può dimostrare che l'algoritmo AdaBoost.M1 (2.2) è equivalente ad una procedura *Forward Stagewise Additive Modeling* in cui viene utilizzata una funzione di perdita esponenziale: $L(y, f(x)) = exp(-y \cdot f(x))$.
Il *gradient boosting* prevede un'ulteriore approssimazione che può essere effettuata per facilitare la risoluzione della 2.4: l'idea è quella di effettuare una discesa del gradiente sulla funzione di perdita in più step. Il primo prevede il calcolo dei gradienti rispetto alle singole N osservazioni in maniera iterativa, mentre il secondo di generalizzare il risulato su tutti i punti dello spazio attraverso la stima di una nuova funzione sulle quantità appena calcolate (algoritmo 2.4).

Il gradient boosting è implementato sulla libreria di R *gbm* (Ridgeway *et al.*, 2006) in maniera abbastanza esaustiva ma non altrettanto efficiente, ragion per cui si è preferito l'utilizzo del più recente *xgboost* di Chen e Guestrin (2016). Quest'ultimo vanta infatti una parallelizzazione della costruzione a livello dei singoli alberi e permette inoltre di aggiungere un termine di regolarizzazione alla funzione di perdita per controllare meglio il sovradattamento.

Algorithm 2.4 Gradient Boosting

1. Inizializzare $f_0(x) = 0$
2. **for** $m = 1, ..., M$ **do:**
   (a) Calcolare il gradiente puntuale della funzione di perdita nella stima corrente:
   $$r_i = -\left[\frac{\partial L(y_i, f_{m-1}(x_i))}{\partial f_{m-1}(x_i)}\right], \qquad i = 1, ..., N$$
   (b) Approssimare l'opposto del gradiente risolvendo
   $$\underset{\gamma}{minimize} \sum_{i=1}^{N} (r_i - f(x_i, \gamma))^2$$
   (c) Aggiornare $f_m(x) = f_{m-1}(x) + \epsilon \cdot f(x, \gamma_m)$, dove ϵ, chiamato anche tasso di apprendimento, è fissato a priori ed ha lo scopo di ponderare l'aggiornamento.
3. Ritornare la sequenza di funzioni $f_m(x), \quad m = 1, ..., M$

2.2 Estrazione delle *features* discriminanti

Quando la dimensione dei dati in input ad un modello sono problematiche e c'è il sospetto che vi sia ridondanza (nelle immagini pixel vicini sono spesso ripetitivi) è conveniente applicare delle forme di riduzione della dimensionalità che spesso si traducono nell'estrazione delle *features* maggiormente rilevanti per la risoluzione del problema in esame. Queste *features* inoltre sono spesso invarianti rispetto a traslazione e a cambiamenti di scala, e quindi possono essere utilizzate per incrementare le performance di quei modelli statistici che non godono di tali proprietà, che nel campo della classificazione di immagini sono essenziali.
Se l'obiettivo è quello di riconoscere immagini di nudo risulta naturale pensare che la caratteristica più discriminante sia la pelle. Storicamente infatti, gli algoritmi di riconoscimento della nudità si sono focalizzati proprio su questa *feature*. Effettuare la classificazione finale basandosi solamente sulla porzione di pelle presente nell'immagine risulta però chiaramente insufficiente: basti pensare a come verrebbe classificata la foto ravvicinata di un volto oppure quella di un gruppo numeroso di persone. In questo senso è utile fornire anche l'informazione relativa alla quantità di pelle appartenente al viso, ragione per la quale nel secondo paragrafo di questa sezione verrà presentato un algoritmo per la localizzazione dei volti.

2.2.1 Riconoscimento della pelle

Esistono due tipi principali di algoritmi per il riconoscimento della pelle: i *pixel-based* e i *region-based*. I primi classificano i pixel individualmente e quindi indipendentemente dai valori dei pixel circostanti, mentre i secondi tengono in considerazione anche quest'ultimi per incrementare le permorfance del modello. Di seguito ci soffermeremo solo sui metodi *pixel-based*, per il semplice motivo che sono gli unici implementati gratuitamente nelle librerie di *computer vision* accessibili su python, il linguaggio di programmazione che è stato utilizzato per la fase di *preprocessing*. Implementare un proprio algoritmo non è stato possibile per via della mancanza di dataset di questo tipo, e crearne uno è improponible dal momento che richiederebbe la classificazione manuale pixel per pixel di ogni immagine.
Costruire un classificatore *pixel-based* significa semplicemente delineare una regione dello spazio colore entro la quale il pixel viene classificato come pelle. La scelta dello spazio colore in particolare risulta fondamentale. L'RGB, che rappresenta il colore dividendolo nelle sue tre componenti principali (rosso verde e blue) non è il più indicato per due ragioni: presenta un'alta correlazione fra i canali e mescola le informazioni di crominanza e luminanza, ossia quelle di maggiore interesse.
Spazi colore più discriminanti per l'identificazione della pelle sono l'HSV e l'YCrCb, i quali possono essere ottenuti entrambi a partire da trasformazione dei canali RGB. L'YCrCb scompone lo spazio colore in luminanza (definita attraverso una somma pesata dei valori RGB) e le componenti di crominanza del rosso (Cr) e del blue (Cb), definite come distanza dal valore di luminanza.

$$YCrCb \quad \begin{cases} Y = 0.299R + 0.587G + 0.114B \\ Cr = R - V \\ Cb = B - V \end{cases}$$

L'HSV invece separa il colore dominante (H) dalla luminosità (V) e dalla saturazione (S), la quale è definita come intensità del colore in relazione alla luminosità. La proprietà più interessante di questo spazio colore è l'invarianza del canale H rispetto a cambiamenti di luminosità da fonti di luce bianca.

$$HSV \quad \begin{cases} H = arcos\dfrac{\frac{1}{2}((R-G)+(R-B))}{\sqrt{((R-G)^2+(R-B)(G-B)}} \\ S = 1 - 3\dfrac{min(R,G,B)}{R+G+B} \\ V = \dfrac{1}{3}(R+G+B) \end{cases}$$

L'obiettivo finale dell'algoritmo consiste nel costruire una regola di decisione in grado di discriminare fra pixel appartenenti alla pelle e non. Ci sono due tipi di approcci che si possono seguire: uno non parametrico e uno parametrico.

Tipicamente l'approccio non parametrico si basa sugli istogrammi: partendo da un dataset sufficientemente rappresentativo, l'idea è quella di dividere lo spazio

colore in una griglia tridimensionale di classi e contare il numero di volte che i pixel classificati come pelle o non pelle cadono all'interno. Una volta ottenuti gli istogrammi 3D, questi vengono normalizzati e convertiti in una distribuzione di probabilità discreta. A questo punto viene applicato un classificatore di Bayes per calcolare la probabilità che il pixel in esame rappresenti la pelle dato il suo vettore c di codifica del colore:

$$P(pelle|c) = \frac{P(c|pelle)P(pelle)}{P(c|pelle)P(pelle) + P(c|\neg pelle)P(\neg pelle)} \tag{2.5}$$

dove $P(c|pelle)$ e $P(c|\neg pelle)$ sono calcolati rispettivamente dagli istogrammi dei pixel di pelle e non pelle, mentre le probabilità a priori $P(pelle)$ e $P(\neg pelle)$ sono stimate a partire dal numero totale di campioni positivi e negativi del training set.

Il vantaggio dei metodi non parametrici basati sugli istogrammi è che sono indipendenti dalla forma della distribuzione del colore della pelle, mentre il più grande limite risiede nel fatto che offrono una scarsa capacità di interpolazione e generalizzazione dei dati. In questo senso i metodi parametrici garantiscono una performance migliore. L'approccio tipico è quello di modellare la distribuzione del colore della pelle attraverso una distribuzione normale multivariata:

$$p(pelle|c) = \frac{1}{2\pi|\Sigma_s|^{1/2}} e^{-\frac{1}{2}(c-\mu_s)'\Sigma_s|^{-1}(c-\mu_s)} \tag{2.6}$$

dove i momenti primo e secondo sono stimati a partire dal *training set*. Adattare una distribuzione unimodale può andar bene se si vuole restringere la classificazione ad una determinata etnia, ma risulta molto limitante se si vogliono cogliere distribuzioni del colore che si concentrano in zone diverse dello spazio di codifica. Di conseguenza è più appropriato adattare una mistura di normali multivariate. La (2.6) diventa quindi

$$p(pelle|c) = \sum_{i=1}^{k} \pi_i \cdot p_i(c|pelle) \tag{2.7}$$

dove k è il numero di componenti della mistura, π_i i relativi pesi ($\sum_{i=1}^{k} \pi_i = 1$), e $p_i(c|pelle)$ le densità delle singole gaussiane multivariate. La stima del modello viene effettuata tramite la tecnica iterativa dell'*Expectation Maximization* (EM), la quale assume che k sia noto a priori. In particolare, la scelta di questo parametro è fondamentale se non si vuole incappare in problemi di sovradattamento. La classificazione finale viene effettuata confrontando la (2.7) con una soglia determinata empiricamente.

2.2.2 Localizzazione dei volti

La maggior parte degli algoritmi esistenti per la localizzazione dei volti considera il riconoscimento facciale come un problema di classificazione binaria. Alcuni di questi

operano direttamente a livello dei singoli pixel senza applicare forme di riduzione della dimensionalità, ma risultano essere molto sensibili alle condizioni di luminosità e alla presenza di rumore. I più diffusi attuano invece una procedura a due stadi: la prima fase prevede l'estrazione di determinate caratteristiche descrittive attraverso l'utilizzo di appositi operatori, mentre la seconda sfrutta tali caratteristiche per costruire il classificatore finale, attraverso l'utilizzo di opportuni modelli come le *Support Vector Machines* o il *Boosting* visto in precedenza. L'algoritmo utilizzato in questa tesi sfrutta l'operatore LBP (*Local Binary Pattern*) per l'estrazione delle caratteristiche, mentre il classificatore finale si basa su AdaBoost (2.2).

Local Binary Pattern

L'idea di usare l'operatore LBP come descrittore facciale è motivata dal fatto che i volti possono essere visti come una composizione di *micro-patterns*, i quali sono in grado di essere riconosciuti molto bene da questo operatore.
L'LBP agisce dividendo l'immagine in regioni e confrontando ogni pixel di ogni regione con l'insieme degli 8 pixel confinanti, uno alla volta e seguendo un senso circolare (orario o antiorario che sia). Quando il valore del pixel preso in esame è maggiore del valore del pixel confinante si registra uno '0', altrimenti un '1'. Al completamento del giro si sarà dunque ottenuto un numero binario composto da 8 cifre, eventualmente convertibile in decimale per convenienza (figura 2.1).

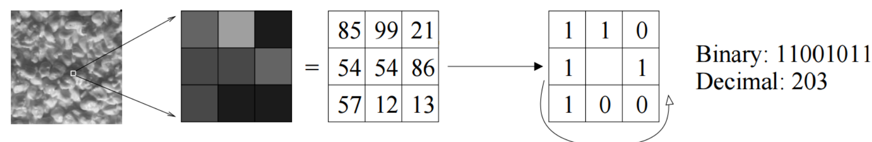


Figura 2.1: Operazione di confronto su un singolo pixel.

Per essere in grado di cogliere *texture* di differenti misure l'LBP può essere esteso utilizzando un sistema di pixel vicini che si espande oltre quelli confinanti. Questo viene ottenuto definendo i pixel vicini (ossia quelli che andranno confrontati con il pixel centrale) come un insieme di P punti equispaziati disposti in un cerchio di raggio R centrato nel pixel preso in esame (figura 2.2). Si fa riferimento a questa struttura con la notazione $LBP_{P,R}$.

La figura 2.3 mostra degli esempi di *preprocessing*, dove i punti neri (o bianchi) rappresentano pixel che sono meno (o più) intensi del pixel centrale. Se i pixel circostanti risultano tutti bianchi o tutti neri significa che compongono una regione di immagine piatta. Gruppi continui di pixel bianchi o neri sono invece considerati pattern uniformi e possono essere interpretati come bordi oppure angoli. Quando invece il pattern registra più di due transizioni da bianco a nero o viceversa, allora

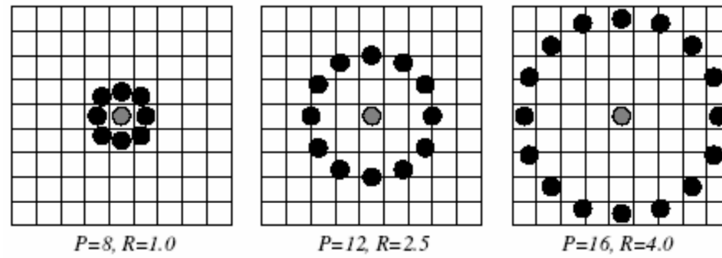


Figura 2.2: Esempi di vicini circolari di diverso raggio (R) e numerosità (P). Il valore dei punti che cadono all'interno di più pixel viene interpolato.

questo viene considerato non uniforme e descrive caratteristiche generalmente più complesse.

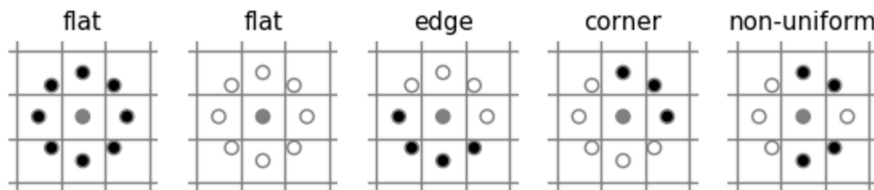


Figura 2.3: Esempi di vicini circolari di diverso raggio (R) e numerosità (P). Il valore dei punti che cadono all'interno di più pixel viene interpolato.

Una volta che tutti i pixel dell'immagine sono stati processati viene prodotto per ogni regione un'istogramma delle frequenze dei numeri ottenuti che descrive una particolare *texture*.

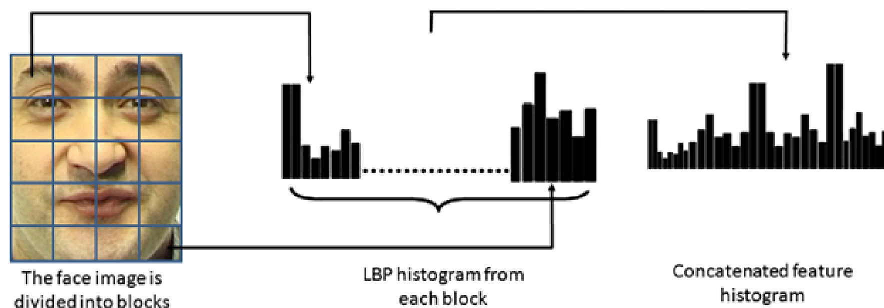


Figura 2.4: L'output finale dell'operatore LBP coincide con l'istogramma ottenuto attraverso la concatenazione degli istogrammi prodotti nelle singole regioni.

Il numero di classi dell'istogramma coincide con il numero di combinazioni ("00100101", "10011101", etc) teoricamente ottenibili, quindi 2^P (256 se $P = 8$). La concatenazione degli istogrammi ottenuti nelle singole regioni produce l'output finale dell'operatore LBP (figura 2.4).

Costruzione del classificatore

L'istogramma che riassume le caratteristiche individuate dall'LBP viene utilizzato nella fase successiva dell'algoritmo per costruire il classificatore finale: servendosi di un apposito dataset contenente esempi positivi e negativi viene stimato un modello AdaBoost (2.2) in cui ogni classificatore debole seleziona la singola classe dell'istogramma che meglio separa le immagini positive da quelle negative.
Il classificatore finale così ottenuto viene applicato su più ritagli della stessa immagine che differiscono per posizione ed estensione, in modo da riuscire a localizzare volti di dimensioni diverse lungo tutta la grandezza dell'immagine.

Capitolo 3

Reti Neurali

Prima di addentrarci nella struttura delle Convolutional Neural Networks è necessario fare un passo indietro richiamando la definizione delle reti neurali classiche, dei modelli ad elevato numero di parametri ispirati all'architettura del cervello umano e promossi come approssimatori universali, ossia dei sistemi che, se alimentati da una sufficiente quantità di dati, sono in grado di apprendere qualsiasi relazione predittiva. Inventate intorno alla metà degli anni '80, le reti neurali hanno goduto di un periodo di popolarità relativamente breve, in quanto sono state presto accantonate da invenzioni più recenti quali il *boosting* e le *support vector machines*. Sono però riemerse attorno al 2010 grazie al notevole miglioramento delle risorse computazionali, decretando così l'inizio del *deep learning* e portando un nuovo entusiasmo nell'ambiente, di fronte a nuove prospettive per la risoluzione di compiti complessi come la classificazione di immagini e video, o il riconoscimento vocale/testuale.

3.1 Motivazione biologica e connessioni

Il neurone è l'elemento basilare del sistema nervoso, ed anche il più importante, tanto che può essere considerato l'unità di calcolo primaria alla base della nostra intelligenza. Il sistema nervoso umano ne è costituito approssimativamente da 86 miliardi, connessi fra di loro da un numero di sinapsi dell'ordine di 10^{15}. Il diagramma sottostante mostra sulla sinistra la rappresentazione biologica di un neurone, mentre sulla destra un comune modello matematico: ogni neurone riceve in input il segnale dai suoi dendriti e, una volta elaborato, produce un segnale di output lungo il suo unico assone, che una volta diramatosi lo collega ai neuroni successivi attraverso le sinapsi. Questa attività biologica può essere rappresentata da un modello matematico nel quale i segnali che viaggiano attraverso gli assoni (gli x_0) interagiscono attraverso un prodotto ($w_0 x_0$) con i dendriti dei neuroni con i quali sono collegati tramite sinapsi (w_0). L'idea è che l'insieme delle sinapsi (i pesi w) possa essere in qualche modo appreso e sia in grado di controllare, in base al segno e all'intensità, l'influenza di un neurone sugli altri. Nel modello base i dendriti portano il segnale al corpo della cellula, dove sono sommati: se il risultato di questa somma supera una

certa soglia, il neurone invierà un impulso attraverso il suo assone. La tempistica degli impulsi non viene considerata rilevante, e si assume pertanto che l'informazione sia portata solamente dalla somma di quest'ultimi. Tale somma viene modellata da quella che viene chiamata funzione di attivazione, che tipicamente coincide con la funzione sigmoide ($\sigma(x) = \frac{1}{1+e^{-x}}$).
In altre parole, ogni neurone esegue un prodotto vettoriale tra i suoi input e il suo set di pesi, somma un termine di distorsione e infine applica una funzione di attivazione non lineare (in caso contrario la rete neurale si ridurrebbe ad un modello lineare generalizzato).

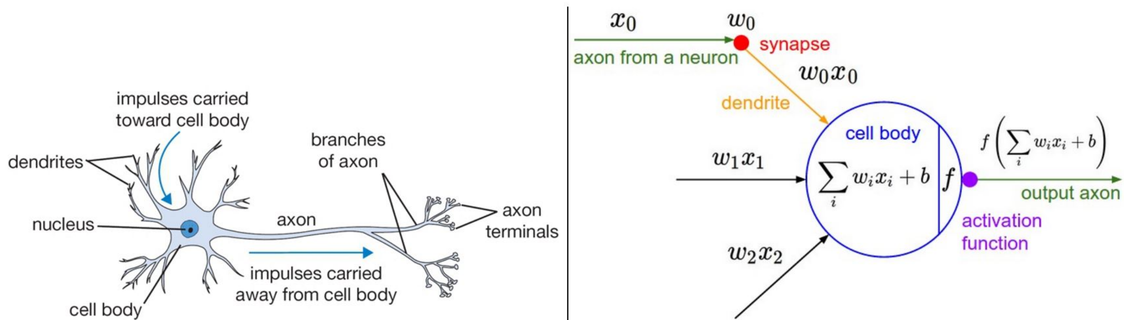


Figura 3.1: Illustrazione biologica di un neurone (sinistra) e rappresentazione del relativo modello matematico (destra). Fonte: Karpathy (2015).

3.2 Architettura di una rete neurale

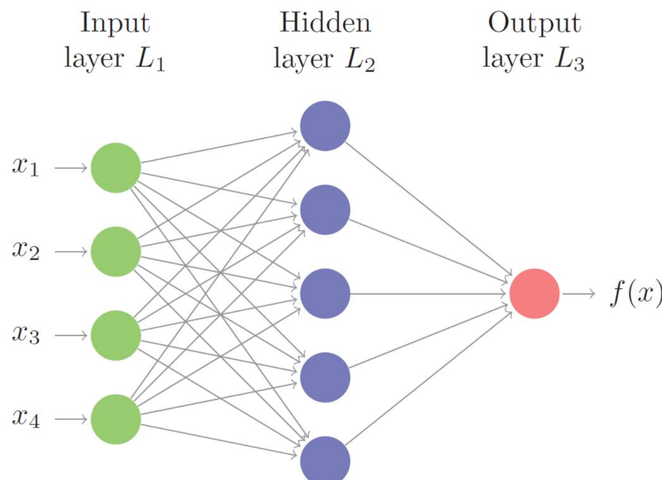


Figura 3.2: Diagramma di una rete neurale con un singolo strato nascosto.

Nella figura 3.2 è rappresentato il diagramma di una semplice rete neurale con 4 predittori x_j, un singolo strato nascosto composto da 5 neuroni $a_l = g(w_{l_0}^{(1)} +$

$\sum_{j=1}^{4} w_{l_j}^{(1)} x_j)$ e una singola unità in uscita $y = h(w_0^{(2)} + \sum_{l=1}^{5} w_l^{(2)} a_l)$. Ogni neurone a_l è connesso allo strato di input attraverso il vettore di parametri o pesi, $\left\{ w_{l_j}^{(1)} \right\}_1^p$ (l'(1) si riferisce al primo strato, l_j alla *j-esima* variabile e *l-esima* unità). I termini d'intercetta w_{l_0} sono necessari per catturare la distorsione, mentre g è la funzione non lineare definita in precedenza. L'idea di base è quella che ogni neurone apprenda una semplice funzione binaria on/off (il lancio dell'impulso), e per questo motivo la funzione g viene anche chiamata funzione di attivazione. Lo strato finale è caratterizzato anch'esso dalla presenza di un vettore di pesi e di una funzione di output h, che generalmente corrisponde alla funzione identità in problemi di regressione, o nuovamente alla funzione sigmoide in problemi di classificazione binaria.

La figura 3.2 rappresenta invece il diagramma di una rete neurale con una struttura più complessa, dove sono presenti tre strati nascosti ed uno strato di output costituito da 10 nodi, uno per ogni possibile classe. Il termine *deep learning* prende nome proprio dalla profondità che caratterizza tali reti, dove con profondità ci si riferisce al numero di strati nascosti presenti. Le reti *"deep"* vengono infatti chiamate anche *MLP*, ovvero "MultiLayer Perceptron".

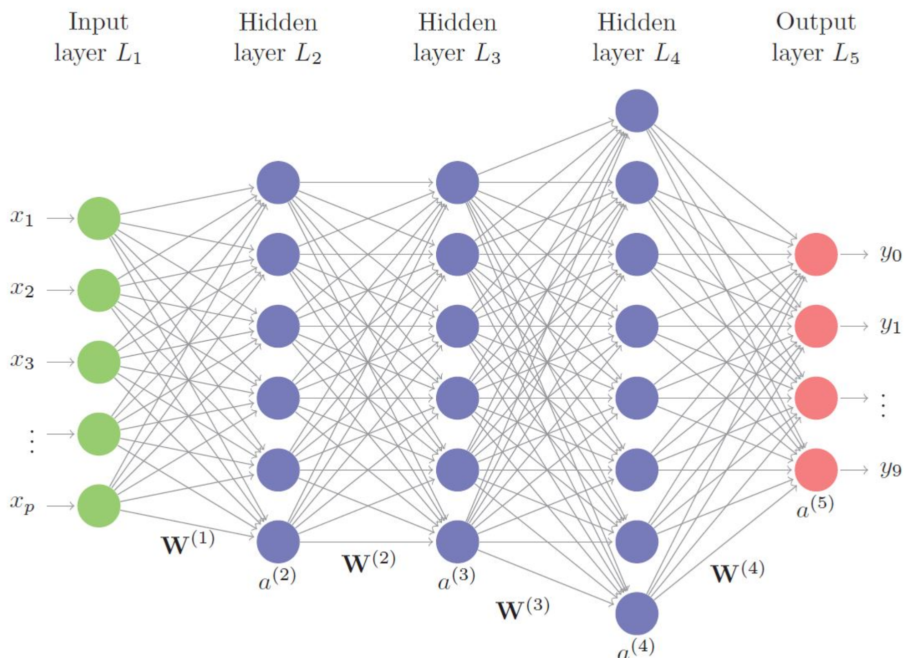


Figura 3.3: Diagramma di una rete neurale con tre strati nascosti e output multipli (Efron e Hastie, 2016).

Utilizzando la notazione introdotta in precedenza possiamo definire il passaggio dallo strato di input al primo strato nascosto come

$$\begin{aligned} z_l^{(2)} &= w_{l_0}^{(1)} + \sum_{j=1}^{p} w_{l_j}^{(1)} x_j \\ a_l^{(2)} &= g^{(2)}(z_l^{(2)}) \end{aligned} \tag{3.1}$$

dove la trasformazione lineare delle x_j è stata separata da quella non lineare delle z_l, input della funzione $g^{(k)}$, la quale non necessariamente deve essere uguale in ogni strato. Più in generale, possiamo definire la transizione dallo strato $k-1$ allo strato k:

$$\begin{aligned} z_l^{(k)} &= w_{l_0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{l_j}^{(k-1)} a_j^{(k-1)} \\ a_l^{(k)} &= g^{(k)}(z_l^{(k)}) \end{aligned} \tag{3.2}$$

o equivalentemente, in termini matriciali

$$\begin{aligned} z^{(k)} &= W^{(k-1)} a^{(k-1)} \\ a^{(k)} &= g^{(k)}(z^{(k)}) \end{aligned} \tag{3.3}$$

dove $W^{(k-1)}$ rappresenta la matrice dei pesi che vanno dallo strato L_{k-1} allo strato L_k, mentre $a^{(k)}$ è il vettore delle attivazioni dello strato L_k. Per problemi di classificazione in M classi (come nel caso della figura 2.2, dove $M = 10$) viene generalmente scelta come trasformazione finale $g^{(K)}$ la funzione *softmax*

$$g^{(K)}(z_m^{(K)}; z^{(K)}) = \frac{e^{z_m^{(K)}}}{\sum_{l=1}^{M} e^{z_l^{(K)}}} \tag{3.4}$$

la quale restituisce una probabilità per ogni classe, con somma lungo le M classi pari ad uno.

3.3 Stima di una rete neurale

Stimare una rete neurale non risulta un compito semplice, trattandosi come abbiamo visto di una complessa funzione gerarchica $f(x; W)$ del vettore di input x e della collezione di pesi W. Inanzitutto è necessario scegliere opportunamente le funzioni di attivazione $g^{(k)}$ in modo che tale funzione risulti differenziabile. Si tratta quindi di risolvere, dato un set di osservazioni $\{x_i, y_i\}_1^n$ e una funzione di perdita $L[y, f(x)]$, un problema di minimizzazione:

$$\underset{W}{min} \left\{ \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i; W)) + \lambda J(W) \right\} \tag{3.5}$$

dove $J(W)$ è una termine non negativo di regolarizzazione sugli elementi di W, con λ relativo parametro di regolazione.

Le funzioni di perdita $L[y, f(x)]$ sono generalmente convesse in f, ma non negli elementi di W, con la conseguenza che ci troviamo a dover risolvere un problema di minimizzazione su una superficie che presenta una grande quantità di minimi locali. Una soluzione è quella di effettuare più stime dello stesso modello con differenti inizializzazioni dei parametri, per scegliere infine quello che risulta essere migliore. Una procedura di questo tipo può però richiedere molto tempo, che spesso non è disponibile, pertanto generalmente ci si tende ad accontentare di buoni minimi locali.

3.3.1 Calcolo del gradiente: l'algoritmo di *back-propagation*

I principali metodi utilizzati per la risoluzione della 3.5 sono basati sulla tecnica della discesa del gradiente, la cui implementazione in questo contesto prende il nome di *back-propagation*, in virtù del fatto che lo scarto registrato in corrispondenza di un certo dato viene fatto propagare all'indietro nella rete per ottenere le formule di aggiornamento dei coefficienti. Dal momento che $f(x; W)$ è definita come una composizione di funzioni a partire dai valori di input della rete, gli elementi di W sono disponibili solo in successione (quella degli strati) e pertanto la differenziazione del gradiente seguirà la regola della catena.
Data una generica osservazione (x, y) l'algoritmo di *back-propagation* prevede di effettuare un primo passo in avanti lungo l'intera rete (*feed-forward* step) e salvare le attivazioni che si creano ad ogni nodo $a_l^{(k)}$ di ogni strato, incluso quello di output. La responsabilità di ogni nodo nella previsione del vero valore di y viene quindi misurata attraverso il calcolo di un termine d'errore $\delta_l^{(k)}$. Nel caso delle attivazioni finali $a_l^{(K)}$ il calcolo di tali errori è semplice: coincide infatti con i residui o loro trasformazioni, a seconda di come viene definita la funzione di perdita. Per le attivazioni degli strati intermedi $\delta_l^{(k)}$ viene calcolato invece come somma pesata dei termini d'errore dei nodi che utilizzano $a_l^{(k)}$ come input.

L'algoritmo 3.1 descrive il calcolo del gradiente della funzione di perdita rispetto alla singola osservazione (x, y). Per il calcolo del gradiente globale sarà sufficiente effettuare la media lungo tutte le osservazioni:

$$\Delta W^{(k)} = \frac{1}{n}\sum_{i=1}^{n} = \frac{\partial L[y, f(x, W)]}{\partial W^{(k)}} \tag{3.9}$$

Una volta calcolato il gradiente è possibile procedere con l'aggiornamento del sistema dei pesi:

$$W^{(k)} \leftarrow W^{(k)} - \alpha\left(\Delta W^{(k)} + \lambda W^{(k)}\right), k = 1, ..., K-1 \tag{3.10}$$

Il calcolo del gradiente ci fornisce la direzione lungo la quale la superficie da minimizzare è più ripida, ma nessuna informazione circa la lunghezza del passo che dovremmo compiere. Tale quantità, rappresentata da α nella 3.10, viene denominata *learning-rate* e costituisce l'iperparametro più importante da regolare in una rete neurale: valori alti portano ad una convergenza più veloce ma sono rischiosi dal

Algorithm 3.1 BACK-PROPAGATION

Requisiti: Derivabilità delle funzioni di attivazione $g^{(k)}$.

1. Data la singola osservazione (x, y) effettuare il passo "*feed-forward*" attraverso il calcolo delle attivazioni $a_l^{(k)}$ in ogni strato $L_2, L_3, ..., L_K$ (calcolare cioè le $f(x; W)$ per ogni x utilizzando il set W di pesi corrente, e tenendo in memoria le quantità calcolate).

2. Per ogni unità l in uscita dallo strato L_K calcolare il termine d'errore:

$$\delta_l^{(K)} = \frac{\partial L[y, f(x, W)]}{\partial z_l^{(K)}} = \frac{\partial L[y, f(x, W)]}{\partial a_l^{(K)}} \dot{g}^{(K)}(z_l^{(K)}) \tag{3.6}$$

   dove $\dot{g}$ denota la derivata di $g(z)$ rispetto a z.

3. Per gli strati $k = K - 1, k - 2, ..., 2$ e per ogni nodo l dello strato k calcolare i termini d'errore intermedi:

$$\delta_l^{(k)} = \left(\sum_{j=1}^{p_{k+1}} w_{jl}^{(k)} \delta_j^{(k+1)} \right) \dot{g}^{(k)}(z_l^{(k)}) \tag{3.7}$$

4. Le derivate parziali sono date da

$$\frac{\partial L[y, f(x, W)]}{\partial w_{lj}^{(k)}} = a_j^{(k)} \delta_l^{(k+1)} \tag{3.8}$$

---

momento che potrebbero saltare il minimo ottimale o fluttuare intorno ad esso, mentre valori bassi sono responsabili di una convergenza lenta e possono comportare il blocco dell'algoritmo in un minimo locale non ottimale.

3.3.2 Stochastic Gradient Descent (SGD)

Esistono alcune varianti dell'algoritmo di discesa del gradiente che differiscono nella quantità di dati utilizzati nel calcolo del gradiente prima di effettuare l'aggiornamento dei parametri. La 3.10 utilizza ad ogni iterazione l'intero dataset, e viene denominata *Vanilla Gradient Descent* o *Batch Gradient Descent*. Tuttavia, può spesso risultare più efficiente processare piccole quantità di dati (*batch*) per volta, generalmente campionate in maniera casuale (da qui il nome *Stochastic Gradient Descent*). Tale scelta è obbligata quando le dimensioni del dataset sono tali da non poter essere caricato in memoria.
Valori estremi del *batch* come n oppure 1 possono causare problemi rispettivamente di calcoli ridondanti (il gradiente viene ricalcolato sempre su osservazioni simili prima dell'aggiornamento) e di fluttuazioni della funzione da minimizzare, a causa di aggiornamenti troppo frequenti e variabili, essendo basati sulla singola osservazione

(figura 3.4). Di conseguenza si tendono a scegliere valori intermedi, tipicamente nel range $[50, 256]$.
Indipendentemente dal numero di iterazioni e dalla dimensione del *batch* prescelta, ogni volta che tutte le n osservazioni del dataset vengono utilizzate per il calcolo del gradiente si dice che viene completata un'epoca (se, come nel *Vanilla Gradient Descent*, la dimensione del *batch* è pari ad n allora ad ogni iterazione corrisponde un'epoca).

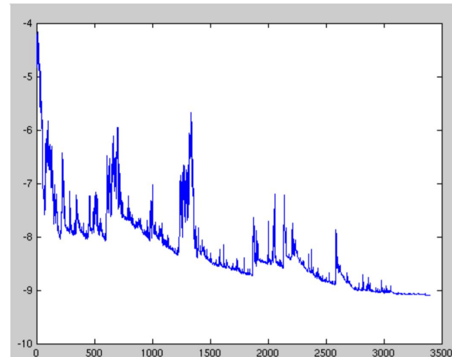


Figura 3.4: Fluttuazioni SGD con dimensione *batch* pari ad uno.

3.3.3 Algoritmi di ottimizzazione della discesa del gradiente

Ci sono alcune modifiche che possono essere apportate all'algoritmo di discesa del gradiente per migliorarne le performance, legate soprattutto alla scelta di α, ovvero il *learning rate*. In aggiunta alle problematiche già discusse in precedenza circa la scelta di questo iperparametro, bisogna sottolineare che nelle configurazioni viste finora viene applicato lo stesso α nell'aggiornamento di tutti i parametri, mentre sarebbe opportuno aggiornare in maniera più significativa quelle caratteristiche (*features*) che risultano meno frequenti, e viceversa.
Altre complicazioni sorgono invece quando si tratta di minimizzare funzione non convesse, con il rischio di rimanere intrappolati in minimi locali non ottimali. Di seguito verranno presentati alcuni algoritmi di ottimizzazione che mirano alla risoluzione di questo tipo di problematiche.

Momentum

Il *Stochastic Gradient Descent* presenta alcune difficoltà in prossimità di aree dove la superficie presenta una curvatura molto più accentuata in una direzione rispetto all'altra. In questo scenario infatti l'algoritmo tende ad oscillare lungo il versante più ripido, rallentando così la convergenza verso il minimo locale ottimale (figura 3.5).

Il *momentum* è un metodo che aiuta ad accelerare la discesa del gradiente verso la direzione corretta, smorzando l'effetto indotto dalle oscillazioni. Tale risultato si

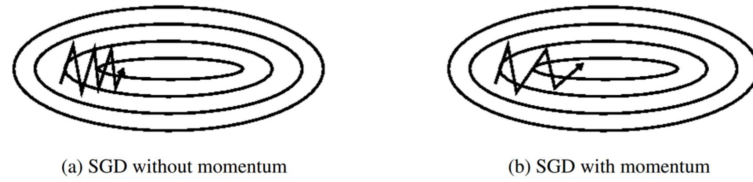


Figura 3.5: Discesa del gradiente senza (a) e con (b) ottimizzazione *momentum*.

ottiene aggiungendo al termine di aggiornamento corrente una frazione γ del vettore di aggiornamento precedente:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_W \varphi(W^{(k)}) \\ W^{(k)} &\leftarrow W^{(k)} - v_t \end{aligned} \tag{3.11}$$

dove $\nabla_W \varphi(W^{(k)})$ è il gradiente rispetto a W della funzione da minimizzare (l'argomento della 3.5). Tipicamente vengono utilizzati valori di γ attorno allo 0.9.

Nesterov Accelerated Gradient

Il Nesterov Accelerated Gradient (NAG) è una variante del *momentum* in cui si cerca di attribuire al termine aggiunto γv_{t-1} una sorta di capacità predittiva. L'idea infatti è che il calcolo di $W^{(k)} - \gamma v_{t-1}$ ci possa fornire un'approssimazione della posizione successiva dei parametri (nel Stochastic Gradient Descent l'aggiornamento non è mai completo). Di conseguenza il calcolo del gradiente non verrà effettuato rispetto al valore corrente dei parametri, ma alla previsione della posizione futura:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_W \varphi(W^{(k)} - \gamma v_{t-1}) \\ W^{(k)} &\leftarrow W^{(k)} - v_t \end{aligned} \tag{3.12}$$

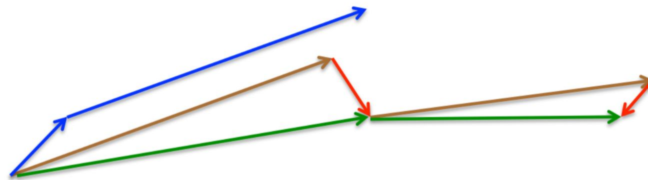


Figura 3.6: Mentre il *momentum* calcola prima il gradiente corrente (freccia piccola blu) per poi saltare in direzione del gradiente accumulato (freccia grande blu), il NAG esegue prima un salto in direzione del gradiente accumulato in precedenza (freccia marrone), e successivamente calcola il gradiente corrente, applicando una correzione (freccia verde).

Adagrad

Momentum e NAG permettono di adattare i nostri aggiornamenti in relazione alla superficie da minimizzare velocizzando così la convergenza, ma non fanno nessuna distinzione circa l'importanza dei parametri, trattandoli tutti allo stesso modo. Adagrad è un algoritmo di ottimizzazione nato proprio con questo scopo: adattare il *learning rate* ai parametri, permettendo così di effettuare aggiornamenti più consistenti in corrispondenza dei parametri relativi alle *features* meno frequenti, e viceversa.
In particolare, nella sua regola di aggiornamento, Adagrad utilizza un α differente per ogni parametro $w_{l_j}^{(k)}$ ad ogni iterazione t, modificando quest'ultimo sulla base dei gradienti passati che sono stati calcolati per lo specifico parametro:

$$w_{t+1,l_j}^{(k)} = w_{t,l_j}^{(k)} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} \nabla_w \varphi(w_{t,l_j}^{(k)}) \tag{3.13}$$

o equivalentemente, in termini matriciali

$$W_{t+1}^{(k)} = W_t^{(k)} - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \nabla_W \varphi(W_t^{(k)}) \tag{3.14}$$

dove $\odot$ indica la moltiplicazione elemento per elemento fra la matrice G_t e il vettore dei gradienti. $G_t \in \mathbb{R}^{d \times d}$ è la matrice diagonale dove ogni elemento (i, i) è la somma dei quadrati dei gradienti calcolati rispetto a $w_{l_j}^{(k)}$ fino all'iterazione t, mentre ϵ è un termine molto piccolo introdotto per evitare eventuali divisioni per zero.
Uno dei più grandi benefici di Adagrad consiste nell'eliminare la necessità di settare manualmente il *learning rate*: è necessario impostare solamente il valore iniziale, tipicamente attorno allo 0.01. D'altra parte, il suo punto debole è invece l'accumulo eccessivo del quadrato dei gradienti al denominatore, che comporta un'esagerata e progressiva riduzione del *learning rate*, il quale tende a diventare infinitamente piccolo, al punto tale che l'algoritmo non è più in grado di acquisire nuova informazione, arrestando così il processo di convergenza.

Adadelta

Adadelta è un'estensione di Adagrad che mira ad alleviare la sua aggressiva e monotona compressione del *learning rate*. Invece di accumulare tutti i gradienti passati, Adadelta ne restringe la finestra di accumulo applicando una media mobile esponenziale. In particolare, la somma dei gradienti viene calcolata ricorsivamente pesando in maniera appropriata valori passati e gradiente corrente:

$$E[\nabla_W \varphi(W^{(k)})^2]_t = \gamma E[\nabla_W \varphi(W^{(k)})^2]_{t-1} + (1 - \gamma) \nabla_W \varphi(W_t^{(k)})^2 \tag{3.15}$$

La 3.14 diventa quindi, in termini di aggiornamento:

$$
\begin{aligned}
\Delta W_t^{(k)} &= -\frac{\alpha}{\sqrt{E[\nabla_W \varphi(W^{(k)})^2]_t + \epsilon}} \nabla_W \varphi(W_t^{(k)}) \\
&= -\frac{\alpha}{RMS[\nabla_W \varphi(W^{(k)})]_t} \nabla_W \varphi(W_t^{(k)})
\end{aligned} \tag{3.16}
$$

Un problema che affligge questa configurazione dell'algoritmo, così come quelle viste finora (SGD, NAG, Adagrad) è che non c'è piena corrispondenza fra le unità di misura dell'aggiornamento ΔW e quelle reali dei pesi W. In altre parole, l'unità di misura di ΔW dipende solo dal gradiente e non dal relativo parametro. Per risolvere questo problema Adadelta implementa una seconda media mobile esponenziale, questa volta non sul quadrato dei gradienti, ma sul quadrato degli aggiornamenti dei parametri:

$$
E[\Delta W^2]_t = \gamma E[\Delta W^2]_{t-1} + (1-\gamma)\Delta W_t^2 \tag{3.17}
$$

con errore quadratico medio

$$
RMS[\Delta W^2]_t = \sqrt{E[\Delta W^2]_t + \epsilon} \tag{3.18}
$$

Dal momento che $RMS[\Delta W^2]_t$ non è disponibile, viene approssimato con l'RMS al passo precedente. Sostituendo infine il *learning rate* con l'$RMS[\Delta W^2]_{t-1}$ otteniamo la regola di aggiornamento finale di Adadelta:

$$
\begin{aligned}
\Delta W_t^{(k)} &= -\frac{RMS[\Delta W^{(k)}]_{t-1}}{RMS[\nabla_W \varphi(W^{(k)})]_t} \nabla_W \varphi(W_t^{(k)}) \\
W_{t+1}^{(k)} &= W_t^{(k)} + \Delta W_t^{(k)}
\end{aligned} \tag{3.19}
$$

Con Adadelta quindi, non è nemmeno necessario impostare un valore iniziale per il *learning rate*, dal momento che questo è stato eliminato dalla regola di aggiornamento.

Adam

Adam, o Adaptive Moment Estimation, è un altro algoritmo che implementa il calcolo adattivo del *learning rate* per ogni parametro. Oltre ad effettuare la media mobile esponenziale del quadrato dei gradienti passati come Adadelta, viene calcolata una media mobile esponenziale anche sul momento primo:

$$
m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_W \varphi(W_t^{(k)}) \tag{3.20}
$$

$$
v_t = \beta_2 v_{t-1} + (1-\beta_2)[\nabla_W \varphi(W_t^{(k)})]^2 \tag{3.21}
$$

Essendo inizializzati a zero, m_t e v_t risultano distorti, in particolare nelle fasi iniziali dell'algoritmo, e soprattutto quando i tassi di decadimento sono bassi (β_1 e β_2 prossimi all'unità), di conseguenza, nella regola di aggiornamento (3.24) viene utilizzata la loro stima corretta:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.22}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.23}$$

$$W_{t+1}^{(k)} = W_t^{(k)} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.24}$$

I valori di default suggeriti dagli autori di Adam (Kingma e Ba, 2014) sono di 0.9 e 0.999 rispettivamente per β_1 e β_2.

Visualizzazione grafica degli algoritmi

Gli algoritmi di ottimizzazione introdotti possono essere messi a confronto visualizzando il percorso che compiono su una determinata superficie di perdita, a partire dallo stesso punto di avvio. Come si può notare in figura 3.7 gli algoritmi seguono tutti un percorso diverso l'uno dall'altro per raggiungere il minimo, con Adagrad e Adadelta che colgono subito la giusta direzione da seguire, mentre *momentum* e NAG tendono per inerzia a scivolare lungo la superficie, correggendo in ritardo la loro traiettoria. Rmsprop (linea nera) è un algoritmo equivalente ad Adadelta, dal quale differisce solo per una specificazione del numeratore leggermente diversa (per approfondimenti si rimanda a [12]).

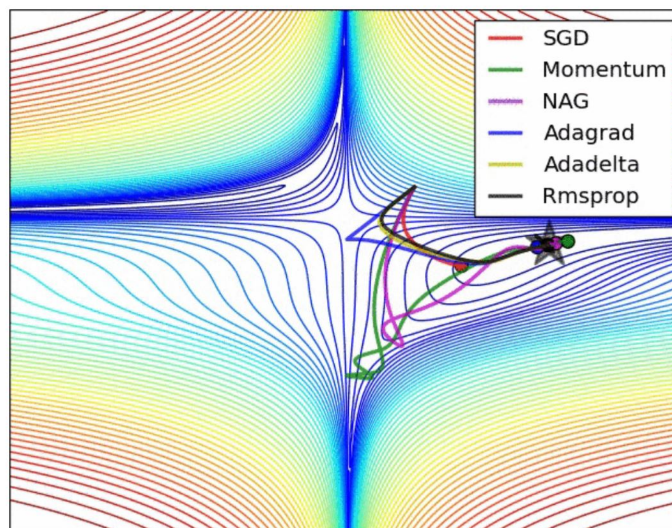


Figura 3.7: Performance degli algoritmi di ottimizzazione SGD sui livelli di una superficie di perdita. Fonte: Ruder (2016).

In figura 3.8 viene mostrato il comportamento degli algoritmi in corrispondenza di un punto di sella, situazione che spesso risulta critica per i metodi di discesa del gradiente. Come menzionato in precedenza, il Stochastic Gradient Descent nella sua versione non ottimizzata mostra qui tutti i suoi limiti, rimanendo incastrato in un moto oscillatorio lungo l'asse con pendenza positiva. La situazione migliora

leggermente per NAG e *momentum*, che per quanto poco riescono almeno a cogliere la giusta direzione da seguire per raggiungere il minimo. Adagrad e Adadelta si dirigono invece subito lungo il versante con pendenza negativa.

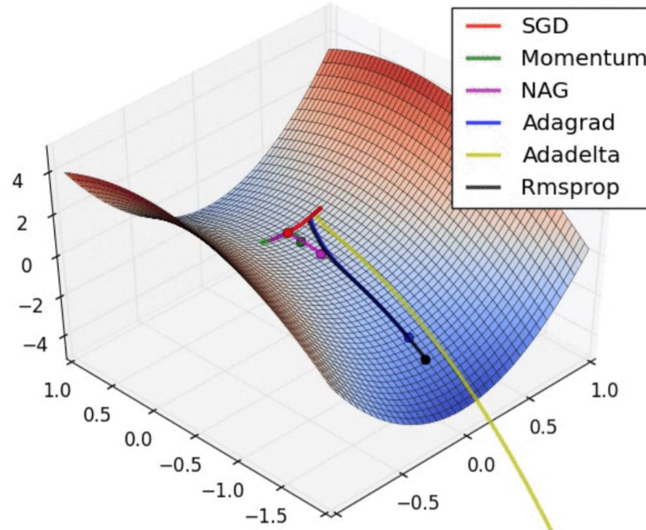


Figura 3.8: Performance degli algoritmi di ottimizzazione SGD su un punto di sella.

3.3.4 Forme di regolarizzazione

In modelli iperparametrizzati come le reti neurali è essenziale ricorrere a delle forme di regolarizzazione in fase di stima, se non si vuole andare incontro a problemi di sovradattamento. L'approccio tipico è quello di inserire un termine di penalizzazione all'interno della funzione di minimizzazione ($J(W)$ nella 3.5) che vanno a regolare l'influenza dei vari parametri all'interno del modello.

Penalizzazioni

Generalmente, il termine di penalizzazione ha una forma del tipo

$$J(W) = \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{l=1}^{p_k - 1} \left\{ \left| w_{l_j}^{(k)} \right| \right\}^q \tag{3.25}$$

dove q è un termine maggiore di zero (se fosse nullo ci troveremo di fronte ad un caso di *subset selection*). A seconda della scelta effettuata per q si va a modificare la forma della regione imposta dal vincolo di penalizzazione (in particolare valori minori di uno rendono la regione convessa, come visibile in figura 2.4).

I casi $q = 1$ e $q = 2$ corrispondono rispettivamente al *lasso* ($L1$) e alla *ridge regression*, nota anche come *weight decay* o ($L2$). La principale differenza fra i due risiede nel fatto che il *lasso*, data la natura del vincolo, tende, all'aumentare di λ, a troncare i pesi a zero, applicando una sorta di selezione continua dei parametri. Un

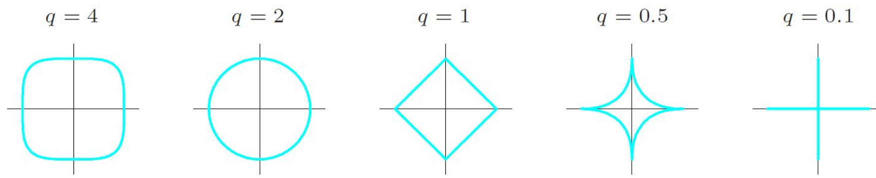


Figura 3.9: Contorni della regione imposta da $J(W) = \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{l=1}^{p_k-1} \left| w_{l_j}^{(k)} \right|^q$, per differenti valori di q.

valido compromesso fra le due forme di penalizzazione è rappresentato dall'*elastic-net*

$$J(W) = \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{l=1}^{p_k-1} \left(\alpha (w_{l_j}^{(k)})^q + (1-\alpha) \left| w_{l_j}^{(k)} \right| \right) \tag{3.26}$$

la quale, presentando una regione con angoli non-differenziabili (Figura 2.5), mantiene la caratteristica del lasso di settare i coefficienti esattamente a zero, ma in maniera molto più moderata, in relazione alla scelta di α.

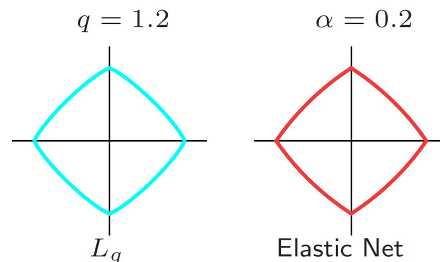


Figura 3.10: Contorni della regione imposta da $J(W) = \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{l=1}^{p_k-1} \left| w_{l_j}^{(k)} \right|^q$ con $q = 2$ (sinistra), elastic-net con $\alpha = 1.2$ (destra). Nonostante i due contorni appaiano simili, solo l'elastic-net risulta non differenziabile negli angoli.

Dropout

Una forma di regolarizzazione alternativa che viene spesso utilizzata nella stima delle reti neurali è il *dropout*, il quale, come suggerisce il nome, agisce "sganciando" in maniera del tutto casuale alcuni nodi, ignorandoli così in fase di stima.

Consideriamo le attivazioni $z_l^{(k)}$ nello strato k per una singola osservazione durante lo stage *feed-forward*: l'idea è quella di settare casualmente a zero ognuno dei p_{k-1} nodi con probabilità ϕ, eliminando temporaneamente dalla rete tutte le sue connessioni, sia quelle in ingresso che quelle in uscita (figura 3.11). Formalmente, la transizione dallo strato $k-1$ allo strato k (3.2) diventa:

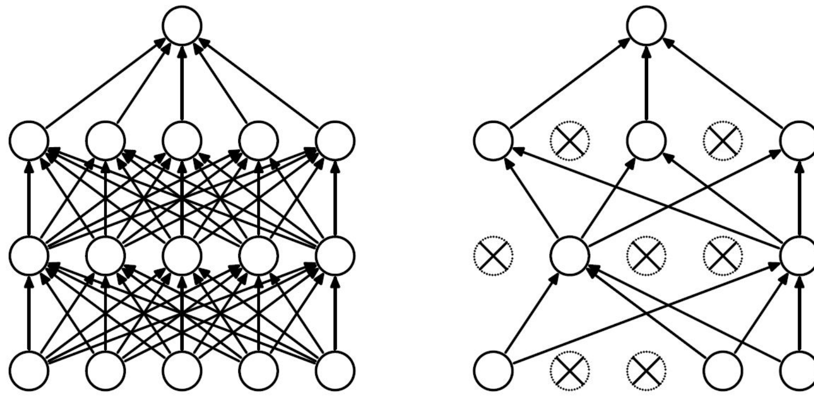


Figura 3.11: Diagramma di una rete neurale prima (sinistra) e dopo (destra) l'applicazione del *dropout*. I nodi segnati con una croce sono quelli che sono stati scelti casualmente per essere rimossi dal modello.

$$\begin{aligned} r_j^{(k-1)} &\sim Bernoulli(1-\phi) \\ \tilde{a}^{(k-1)} &= a^{(k-1)} \odot r_j^{(k-1)} \\ z_l^{(k)} &= w_{l_0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{l_j}^{(k-1)} \tilde{a}_j^{k-1} \\ a_l^{(k)} &= g^{(k)}(z_l^{(k)}) \end{aligned} \tag{3.27}$$

Questo processo può essere applicato su più strati con valori di ϕ non necessariamente uguali (tipicamente vengono utilizzati valori maggiori negli strati più densi e/o finali, lasciando intatto lo strato di input).

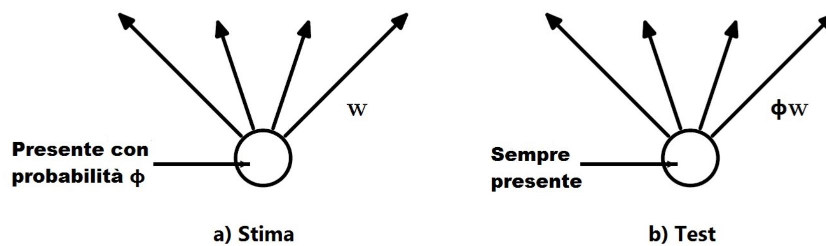


Figura 3.12: Sinistra: il neurone durante la fase di stima è presente con probabilità ϕ, ed è connesso con i neuroni dello strato successivo attraverso i pesi w. Destra: lo stesso neurone in fase di test è sempre presente nella rete e i suoi pesi sono moltiplicati per ϕ.

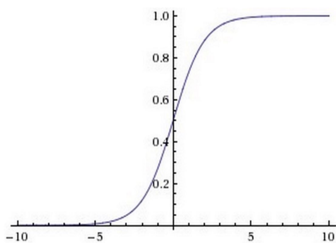
L'applicazione del *dropout* produce quindi ad ogni iterazione (stage *feed-forward* delle singole osservazioni) una diversa rete ridotta del modello di partenza, composta da quei nodi che sono sopravvissuti al (temporaneo) processo di eliminazione. Di conseguenza viene a crearsi una sorta di combinazione di modelli che tende quasi

sempre a migliorare la performance finale, riducendo in particolare l'errore di generalizzazione. Ovviamente in fase di test, mediare tutte le reti ridotte per ottenere una previsione diventa improponibile per ovvie ragioni di tempo, specialmente in quei contesti in cui c'è la necessità di avere una risposta quasi istantanea. L'idea quindi, è quella di utilizzare una singola rete neurale completa, i cui pesi sono le versioni ridimensionate dei pesi calcolati in precedenza: se un neurone aveva una probabilità ϕ di essere eliminato dal modello in fase di stima, allora i suoi pesi in uscita verranno moltiplicati per ϕ (figura 3.12).

3.4 Scelte di non linearità

3.4.1 Funzione Sigmoide

Fino ad ora si è parlato solamente della sigmoide come scelta per la funzione di attivazione non lineare, ma tale soluzione è stata progressivamente accantonata negli ultimi anni per via di alcune problematiche che comporta a livello pratico.



$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{3.28}$$

La prima e più importante è quella relativa alla dissolvenza del gradiente in seguito alla saturazione dei neuroni, ossia quei neuroni che presentano valori di output agli estremi del codominio della funzione di attivazione, in questo caso $(0, 1)$. É facile infatti notare che

$$\begin{aligned} \sigma(z)_{z \to \infty} &= 1 \\ \sigma(z)_{z \to -\infty} &= 0 \end{aligned} \tag{3.29}$$

Tale saturazione diventa problematica durante le fase di *back propagation*, in quanto il gradiente locale assume valori prossimi allo zero (3.30), che per la regola della catena vanno a moltiplicare tutti i gradienti calcolati in precedenza e quelli successivi, conducendo così all'annullamento del gradiente globale.

$$\begin{aligned} \sigma^{'}(z) &= \sigma(z)(1-\sigma(z)) \\ \sigma^{'}(z)_{z \to \infty} &= 0 \\ \sigma^{'}(z)_{z \to -\infty} &= 0 \end{aligned} \tag{3.30}$$

In pratica, si ha un flusso utile del gradiente solo per valori di input che rimangono all'interno di una zona di sicurezza, cioè nei dintorni dello zero.
Il secondo problema deriva invece dal fatto che gli output della funzione sigmoide non sono centrati intorno allo zero, e di conseguenza gli strati processati successivamente

riceveranno anch'essi valori con una distribuzione non centrata sullo zero. Questo influisce in maniera significativa sulla discesa del gradiente, in quanto gli input in ingresso ai neuroni saranno sempre positivi, e pertanto il gradiente dei pesi associati diventerà, durante la fase di *back propagation*, sempre positivo o sempre negativo. Tale risultato si traduce in una dinamica a zig-zag negli aggiornamenti dei pesi che rallenta in maniera significativa il processo di convergenza.

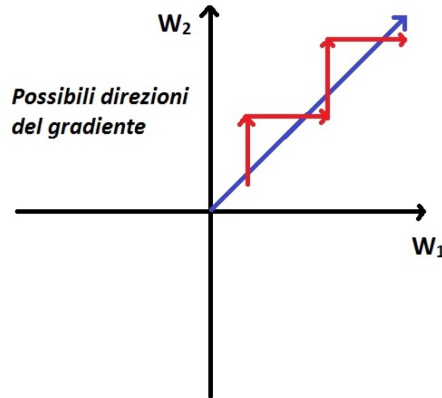
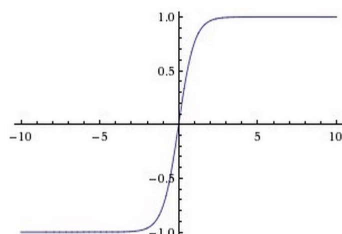


Figura 3.13: Esempio di dinamica a zig-zag nell'aggiornamento di due pesi W_1 e W_2. La freccia blu indica indica l'ipotetico vettore ottimale per la discesa del gradiente, mentre le frecce rosse i passi di aggiornamento compiuti: gradienti tutti dello stesso segno comportano due sole possibili direzioni di aggiornamento.

É importante comunque notare che una volta che i gradienti delle singole osservazioni vengono sommati all'interno dello stesso *batch* di dati l'aggiornamento finale dei pesi può avere segni diversi, permettendo quindi di muoversi lungo un insieme più ampio di direzioni.
Il terzo e ultimo difetto della funzione sigmoide è che l'operazione $exp(\cdot)$ al denominatore è molto costosa dal punto dal punto di vista computazionale, soprattutto rispetto alle alternative che verranno presentate di seguito.

3.4.2 Tangente iperbolica

Il problema degli output non centrati sullo zero della sigmoide può essere risolto ricorrendo all'utilizzo della tangente iperbolica, la quale presenta codominio $(-1, 1)$ centrato sull'origine degli assi.

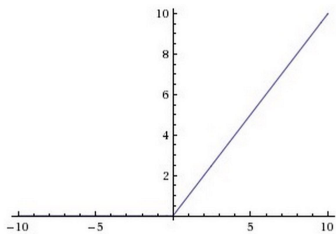


$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{3.31}$$

Tuttavia, rimane il problema della saturazione dei neuroni, anzi viene addirittura accentuato, dal momento che la zona di sicurezza risulta ancora più ristretta.

3.4.3 ReLU

La Rectified Linear Unit (ReLU) è diventata popolare negli ultimi anni per via dell'incremento prestazionale che offre nel processo di convergenza: velocizza infatti di circa 6 volte la discesa del gradiente rispetto alle alternative viste finora.



$$ReLU(z) = max(0, z) \tag{3.32}$$

Questo risultato è da attribuire in larga parte al fatto che la ReLU risolve il problema della dissolvenza del gradiente, non andando a saturare i neuroni. Durante la fase di *back propagation* infatti, se il gradiente calcolato fino a quel punto è positivo questo viene semplicemente lasciato passare, perchè la derivata locale per il quale viene moltiplicato è pari ad uno (formula 3.33).

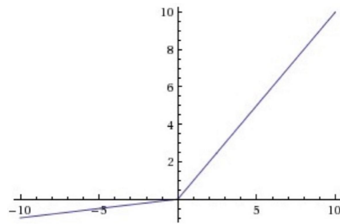
$$\frac{ReLU(z)}{\partial z} = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases} \tag{3.33}$$

Eventuali problemi sorgono invece quando il gradiente accumulato ha segno negativo, in quanto questo viene azzerato (le derivata locale è nulla lungo tutto il semiasse negativo) con la conseguenza che i pesi non vengono aggiornati. Fortunatamente questo problema può essere alleviato attraverso l'utilizzo di un algoritmo SGD (3.3.2): considerando più dati alla volta c'è infatti la speranza che non tutti gli input del *batch* provochino l'azzeramento del gradiente, tenendo così in vita il processo di apprendimento del neurone. Al contrario, se per ogni osservazione la ReLU riceve valori negativi, allora il neurone "muore", e non c'è speranza che i pesi vengano aggiornati. Valori elevati del *learning rate* amplificano questo problema, dal momento che cambiamenti più consistenti dei pesi si traducono in una maggiore probabilità che questi affondino nella "zona morta".

3.4.4 Leaky ReLU

La leaky-ReLU è un tentativo di risolvere il problema della disattivazione dei neuroni comportato dalla ReLU classica, e consiste nell'introdurre una piccola pendenza negativa (α) nella regione dove la ReLU è nulla:

dove α è una costante. In alcune varianti, α può essere un parametro da stimare, al pari degli altri pesi della rete (si parla di *Parametric ReLU*), oppure una variabile casuale: è il caso della *Randomized ReLU*, dove ad ogni iterazione la pendenza



$$leakyReLU(z) = max(\alpha z, z)$$

della parte negativa della funzione viene scelta casualmente all'interno di un range prefissato.
In alcuni recenti esperimenti, Bing Xu *et al.* (2015) hanno mostrato come le varianti della ReLU classica siano in grado di aumentare le performance finali del modello in termini di accuratezza, prima su tutte la RReLU, che grazie alla sua natura casuale sembra particolarmente portata alla riduzione del sovradattamento.

3.5 Inizializzazione dei pesi

Un passo molto delicato durante la fase di stima di una rete neurale è quello relativo all'inizializzazione dei pesi. Se da un lato ci risulta naturale pensare che i valori iniziali non siano rilevanti (dal momento che è compito dei successivi aggiornamenti farli convergere in direzione dei valori cercati), dall'altro lato è anche vero che, viste le problematiche legate alle funzioni di attivazione, la rete neurale non è più in grado di apprendere se si va incontro alla dissolvenza del gradiente.
Consideriamo alcuni casi in cui la funzione di attivazione scelta è la tangente iperbolica vista in precedenza; Impostare i pesi a zero non ha alcun senso, dal momento che tutti i nodi produrrebbero lo stesso identico output, quindi generalmente questi vengono inizializzati attraverso generazioni casuali da una variabile normale a media nulla e varianza molto piccola (0.01). Questo tipo di inizializzazione può andare bene se la rete neurale dispone di pochi strati nascosti, ma in reti neurali più profonde le attivazioni diventano sempre più piccole mano a mano che si processano i vari strati, fino ridursi a quantità praticamente nulle. Questo diventa un problema in quanto durante la fase di *back propagation* il gradiente accumulato continua ad essere moltiplicato per quantità piccolissime che portano alla sua dissolvenza.
D'altra parte, inizializzazioni con valori più grandi (ad esempio da normali standard) conducono al problema già visto della saturazione dei neuroni, e di conseguenza nuovamente alla dissolvenza del gradiente con relativo arresto del processo di aggiornamento dei pesi.
L'idea è quindi quella di avere una distribuzione delle attivazioni tale che la rete neurale sia in grado di apprendere in maniera efficiente. In quest'ottica, Xavier (2010) ha proposto una inizializzazione dei pesi secondo una normale con deviazione standard tale che la varianza delle attivazioni $a^{(k)}$ risulti essere unitaria. Sotto l'assunzione di attivazioni lineari (plausibile dal momento che la tangente iperbolica ha proprio questo comportamento intorno allo zero) questo si traduce nel rendere unitaria la varianza degli input $z^{(k)}$:

$$
\begin{aligned}
Var(z_l^{(k)}) &= Var\left(\sum_{j=1}^{p_{k-1}}\left(w_{l_j}^{(k-1)}a_j^{(k-1)}\right)\right) \\
&= \sum_{j=1}^{p_{k-1}}(E[w_{l_j}^{(k-1)}]^2 Var(a_j^{(k-1)}) + E[a_j^{(k-1)}]^2 Var(w_{l_j}^{(k-1)}) \\
&\quad + Var(w_{l_j}^{(k-1)})Var(a_j^{(k-1)}))
\end{aligned}
\tag{3.34}
$$

Se si assume che le attivazioni siano distribuite simmetricamente intorno allo zero e ci sia indipendenza con i pesi, la 3.34 diventa

$$
\begin{aligned}
Var(z_l^{(k)}) &= \sum_{j=1}^{p_{k-1}} Var(w_{l_j}^{(k-1)})Var(a_j^{(k-1)}) \\
&= p_{k-1}Var(w_{l_j}^{(k-1)})
\end{aligned}
\tag{3.35}
$$

Di conseguenza, affinchè gli input z_l abbiano varianza unitaria dovremmo inizializzare i pesi w_{l_j} secondo una normale a media nulla e varianza pari all'inverso del numero di connessioni in ingresso:

$$
w_{l_j}^{(k-1)} \sim N\left(0, \frac{1}{p_{k-1}}\right) \tag{3.36}
$$

La 3.36 viene chiamata inizializzazione di *Xavier* e in alcuni esperimenti *He et al. (2015)* hanno dimostrato che sembra funzionare a dispetto delle numerose assunzioni su cui si basa.
Per la ReLU il passaggio dalla 3.34 alla 3.35 non vale, in quanto viene violata l'assunzione di simmetria attorno allo zero. *He et al. (2015)* suggeriscono in questo caso di generare i pesi da una normale a media nulla e varianza doppia rispetto alla 3.36:

$$
w_{l_j}^{(k-1)} \sim N\left(0, \frac{2}{p_{k-1}}\right) \tag{3.37}
$$

Il fattore di correzione 2 ha senso: la ReLU dimezza di fatto gli input, e pertanto bisogna raddoppiare la varianza dei pesi per mantenere la stessa varianza delle attivazioni.

3.5.1 Batch Normalization

Nel 2015, Szegedy *et al.* (2015) hanno proposto una soluzione più semplice ed intuitiva al problema del cambiamento nella distribuzione degli input, ossia quella di

andarli a normalizzare direttamente. Dati $z_{l|i}^{(k)}$ input dello strato k generati dall'*i-esima* osservazione, la procedura ideale prevederebbe di sbiancare congiuntamente gli input lungo l'intero dataset $\mathcal{X}$. Una procedura di questo tipo risulta però molto costosa dal momento che richiede il calcolo dell'intera matrice di covarianza $Cov[z] = E_{z \in \mathcal{X}}[zz^T] - E[z]E[z]^T$. Inoltre, potrebbe creare problemi durante la fase di *back propagation* non garantendo la differenziabilità della funzione di perdita. Per questi motivi viene privilegiato un approcio semplificato, in grado di normalizzare gli input in una maniera differenziabile e che non richiede l'analisi dell'intero dataset.

La prima di queste semplificazioni consiste nel normalizzare in maniera indipendente ogni nodo; in altre parole, dato un vettore di input z con p_k elementi, $z = (z_1, ..., z_{p_k})$, si tratta di normalizzare ogni componente l:

$$\hat{z}_l^{(k)} = \frac{z_l^{(k)} - E[z_l^{(k)}]}{\sqrt{Var[z_l^{(k)}]}} \tag{3.38}$$

dove media e varianza sono calcolate lungo le diverse osservazioni del dataset.
A fronte di una soluzione così semplice sorge però spontaneo chiedersi se cambiare manualmente la distribuzione degli input non modifica ciò che lo strato può rappresentare. Per porre rimedio a questo problema vengono introdotti per ogni input $z_l^{(k)}$ due parametri $\gamma_l^{(k)}$ e $\beta_l^{(k)}$, i quali hanno il compito di scalare e traslare i valori normalizzati:

$$t_l^{(k)} = \gamma_l^{(k)} \hat{z}_l^{(k)} + \beta_l^{(k)} \tag{3.39}$$

$\gamma_l^{(k)}$ e $\beta_l^{(k)}$ vengono stimati assieme agli altri parametri del modello, e rispristinano la capacità rappresentativa della rete. Infatti, impostando $\gamma_l^{(k)} = \sqrt{Var[z_l^{(k)}]}$ e $\beta_l^{(k)} = E[z_l^{(k)}]$ si ottengono gli input originali $z_l^{(k)}$.
Una seconda semplificazione viene eseguita in virtù del fatto che per ragioni computazionali nella pratica viene effettuata quasi sempre un'ottimizzazione SGD (3.3.2), la quale non processa l'intero dataset ma *batch* di dati per volta. Di conseguenza anche l'operazione di normalizzazione degli input verrà effettuata per ogni *batch*. Di seguito viene riassunto l'algoritmo di normalizzazione, in cui per semplicità ci si focalizzerà solo su un singolo input l dello strato k ($z_l^{(k)}$), in particolare sul relativo vettore di m valori $z_{l|i}^{(k)}$ generato dalle m osservazioni del *batch* corrente:

$$\mathcal{B} = \left\{ z_{l|1}^{(k)}, ..., z_{l|m}^{(k)} \right\} \tag{3.40}$$

Algorithm 3.2 BATCH NORMALIZATION

Input: Valori $z_{l|i}^{(k)}$ generati dal batch corrente: $\mathcal{B} = \left\{ z_{l|1}^{(k)}, ..., z_{l|m}^{(k)} \right\}$;

Output: $\left\{ t_{l|i}^{(k)} = BN_{\gamma_l^{(k)}, \beta_l^{(k)}}(z_{l|i}^{(k)}) \right\}$;

1. $\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} z_{l|i}^{(k)}$ // media del *batch*
2. $\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} \left(z_{l|i}^{(k)} - \mu_{\mathcal{B}} \right)^2$ // varianza del *batch*
3. $\hat{z}_{l|i}^{(k)} \leftarrow \frac{z_{l|i}^{(k)} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ // normalizzazione
4. $t_{l|i}^{(k)} \leftarrow \gamma_l^{(k)} \hat{z}_{l|i}^{(k)} + \beta_l^{(k)} \equiv BN_{\gamma_l^{(k)}, \beta_l^{(k)}}(z_{l|i}^{(k)})$ // trasformazione lineare

Durante la fase stima della rete c'è bisogno di propagare all'indietro il gradiente della funzione di perdita f attraverso la trasformazione 3.39, e in particolare di calcolare le derivate rispetto a γ e β, dato che, come detto in precedenza, rientrano anch'essi a tutti gli effetti fra i parametri da stimare. I passi sono quelli soliti dettati dalla regola della catena:

$$
\begin{aligned}
\frac{\partial f}{\partial \hat{z}_{l|i}^{(k)}} &= \frac{\partial f}{\partial t_{l|i}^{(k)}} \cdot \gamma_l^{(k)} \\
\frac{\partial f}{\partial \sigma_{\mathcal{B}}^2} &= \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{z}_{l|i}^{(k)}} \cdot (z_{l|i}^{(k)} - \mu_{\mathcal{B}}) \cdot \left(-\frac{1}{2} \right) (\sigma_{\mathcal{B}}^2 + \epsilon)^{-\frac{3}{2}} \\
\frac{\partial f}{\partial \mu_{\mathcal{B}}} &= \left(\sum_{i=1}^{m} \frac{\partial f}{\partial \hat{z}_{l|i}^{(k)}} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \right) + \frac{\partial f}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(z_{l|i}^{(k)} - \mu_{\mathcal{B}})}{m} \\
\frac{\partial f}{\partial z_{l|i}^{(k)}} &= \frac{\partial f}{\partial \hat{z}_{l|i}^{(k)}} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial f}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(z_{l|i}^{(k)} - \mu_{\mathcal{B}})}{m} + \frac{\partial f}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m} \\
\frac{\partial f}{\partial \gamma_l^{(k)}} &= \sum_{i=1}^{m} \frac{\partial f}{\partial t_{l|i}^{(k)}} \cdot \hat{z}_{l|i}^{(k)} \\
\frac{\partial f}{\partial \beta_l^{(k)}} &= \sum_{i=1}^{m} \frac{\partial f}{\partial t_{l|i}^{(k)}}
\end{aligned}
\tag{3.41}
$$

L'algoritmo di 3.2 rappresenta quindi una trasformazione differenziabile in grado di normalizzare gli input senza creare problemi durante la fase di *back propagation*.
Il fatto che il processo di normalizzazione dipenda solamente dal *batch* corrente e non dall'intero dataset non è però una proprietà desiderabile dal punto di vista inferenziale, piuttosto un compromesso che si accetta per rendere efficiente l'allenamento della rete. Per questo motivo, in fase di test, gli input vengono normalizzati sulla base delle statistiche relative all'intera popolazione:

$$\hat{z} = \frac{z - E[z]}{\sqrt{Var[z] + \epsilon}} \tag{3.42}$$

dove $Var[x] = \frac{m}{m-1} \cdot E_{\mathcal{B}}[\sigma^2_{\mathcal{B}}]$, con valore atteso calcolato lungo tutti i *batches* di dimensione m.
L'algoritmo 3.3 riassume il processo di *training* di una rete neurale con normalizzazione degli input.

Algorithm 3.3 TRAINING DI UNA RETE BN

Input: Rete N e relativo set di parametri Θ;
Subset di input $\left\{z_l^{(k)}\right\}_{l=1}^{p_k}$

Output: Rete *"Batch Normalized"* per l'inferenza: N_{BN}^{inf}

1: $N_{BN}^{tr} \leftarrow N$
2: **for** $l = 1, ..., p_k$ **do**
3: $t_l^{(k)} = BN_{\gamma_l^{(k)}, \beta_l^{(k)}}(z_l^{(k)})$ // Algoritmo 3.2
4: Modificare ogni strato di N_{BN}^{tr} sostituendo gli input $z_l^{(k)}$ con quelli normalizzati $t_l^{(k)}$;
5: **end for**
6: Allenare la rete N_{BN}^{tr} per ottimizzare i parametri $\Theta \cup \left\{\gamma_l^{(k)}, \beta_l^{(k)}\right\}_{l=1}^{p_k}$
7: $N_{BN}^{inf} \leftarrow N_{BN}^{tr}$
8: **for** $l = 1, ..., p_k$ **do**
9: // Per semplicità, $z \equiv z_l^{(k)}, \gamma \equiv \gamma_l^{(k)}, \beta \equiv \beta_l^{(k)}$
10: Processare tutti i *batch* $\mathcal{B}$ di training, ognuno di dimensione m, e calcolare i momenti dell'intera popolazione:

$$E[z] \leftarrow E_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$Var[z] \leftarrow \frac{m}{m-1} \cdot E_{\mathcal{B}}[\sigma^2_{\mathcal{B}}]$$

11: Nella rete N_{BN}^{inf}, sostituire i valori di $t = BN_{\gamma,\beta}(z)$ con

$$t = \frac{\gamma}{\sqrt{Var[z] + \epsilon}} \cdot z + \left(\beta - \frac{\gamma E[z]}{\sqrt{Var[z] + \epsilon}}\right)$$

12: **end for**

Avere una rete con input normalizzati rappresenta un vantaggio indiscusso, in quanto previene il pericoloso processo di dissolvenza del gradiente e in particolare la saturazione dei neuroni. Oltre ad ottenere una stima più efficiente questo ci permette di utilizzare valori più alti del *learning rate* (che senza normalizzazione andrebbero a saturare i neuroni) e di conseguenza accelerare il processo di convergenza.
Altro punto a favore della *Batch Normalization* è che agisce involontariamente come forma di regolarizzazione del modello. Durante la fase di stima infatti, le attivazioni

che vengono calcolate non sono più un prodotto deterministico delle singole osservazioni, in quanto durante il processo di normalizzazione queste vengono valutate congiuntamente alle altre osservazioni del *batch*. Secondo Szegedy *et al.* (2015) il guadagno in termini di generalizzazione è tale da ridurre di un fattore pari a 5 l'entità del *weight decay*, o di rimuovere il *dropout* senza comportare un aumento del sovradattamento.

Capitolo 4

Convolutional Neural Networks

Le *Convolutional Neural Networks* (alle quali si farà riferimento da ora in poi, per brevità, con l'acronimo CNN) sono reti neurali specializzate nel processamento di dati che presentano una struttura a griglia. Alcuni esempi sono le serie storiche (che possono essere pensate come una griglia monodimensionale di campionamento ad intervalli di tempo regolari), le immagini in bianco e nero (griglia bidimensionale dove il singolo valore rappresenta l'intensità del pixel in scala di grigi) o come nel caso di questa tesi, immagini a colori, dove la griglia questa volta è tridimensionale perchè concatena lungo la terza dimensione le griglie bidimensionali relative ai singoli canali colore attraverso i quali l'immagine è codificata (tipicamente RGB - Red Green Blue).
Il nome *Convolutional Neural Networks* deriva dal fatto che tali reti utilizzano un'operazione matematica lineare chiamata convoluzione. Di conseguenza, una rete neurale classica che implementa operazioni di convoluzione in almeno uno dei suoi strati, viene definita *Convolutional*.
Gli strati composti da operazioni di convoluzione prendono il nome di *Convolutional Layers*, ma non sono gli unici strati che compongono una CNN: la tipica architettura prevede infatti l'alternarsi di *Convolutional Layers*, *Pooling Layers* e *Fully Connected Layers*.

Inadeguatezza della struttura ***fully-connected***

Come visto nel capitolo precedente, le reti neurali tradizionali ricevono in input un singolo vettore, e lo trasformano attraverso una serie di strati nascosti, dove ogni neurone è connesso ad ogni singolo neurone sia dello strato precedente che di quello successivo (ovvero "*fully-connected*") e funziona quindi in maniera completamente indipendente, dal momento che non vi è alcuna condivisione delle connessioni con i nodi circostanti. Nel caso l'input sia costituito da immagini di dimensioni ridotte, ad esempio $32 \times 32 \times 3$ (32 altezza, 32 larghezza, 3 canali colore), un singolo neurone connesso in questa maniera comporterebbe un numero totale di $32 \times 32 \times 3 = 3072$ pesi, una quantità abbastanza grande ma ancora trattabile. Le cose si complicano però quando le dimensioni si fanno importanti: salire ad appena 256 pixel per la-

to comporterebbe un carico di $256 \times 256 \times 3 = 196\,608$ pesi per singolo neurone, ovvero quasi 2 milioni di parametri per una semplice rete con un singolo strato nascosto da dieci neuroni. L'architettura *fully-connected* risulta perciò troppo esosa in questo contesto, comportando una quantità enorme di parametri che condurrebbe velocemente a casi di sovradattamento. Inoltre, considerazione ancor più limitante, un'architettura di questo tipo fatica a cogliere la struttura di correlazione tipica dei dati a griglia.
Le *Convolutional Neural Networks* prendono invece vantaggio dall'assunzione che gli input hanno proprio una struttura di questo tipo, e questo permette loro la costruzione di un'architettura su misura attraverso la formalizzazione di tre fondamentali proprietà: l'interazione sparsa (***sparse interaction***), l'invarianza rispetto a traslazioni (***invariant to translation***), e la condivisione dei parametri (***weight sharing***). Il risultato è una rete più efficace e allo stesso tempo parsimoniosa in termini di parametri.

4.1 L'operazione di convoluzione

In problemi discreti l'operazione di convoluzione non è altro che la somma degli elementi del prodotto di Hadamard fra un set di parametri (che prende il nome di filtro) e una porzione dell'input di pari dimensioni. La figura 4.1 aiuta a comprendere meglio questa operazione in realtà molto semplice, attraverso un esempio nel caso di un input bidimensionale.

L'operazione di convoluzione viene quindi ripetuta spostando il filtro lungo tutta la superficie dell'input, sia in altezza che in larghezza. Questo produce quella che viene chiamata mappa di attivazioni (o *features map*, figura 4.2), la quale costituisce di fatto il primo strato nascosto della rete.

In altre parole, ragionando dalla prospettiva opposta, la mappa di attivazioni è formata da neuroni connessi localmente allo strato di input attraverso i parametri del filtro che li ha generati. L'estensione spaziale di questa connettività, che coincide con la dimensione del filtro, costituisce un iperparametro della rete e viene chiamata anche campo recettivo del neurone, o *receptive field*. Questa rappresenta la prima delle tre proprietà fondamentali delle *Convolutional Neural Networks*, ossia la ***sparse interaction***.
Formalmente, riprendendo la notazione del capitolo 3, la transizione dallo strato di input al primo strato nascosto diventa:

$$\begin{aligned} z^{(2)}_{(i,j)} &= w^{(1)}_0 + \sum_{a=1}^{F} \sum_{b=1}^{F} w^{(1)}_{(a,b)} x_{(i+a-1,j+b-1)} \\ a^{(2)}_{(i,j)} &= g^{(2)}\left(z^{(2)}_{(i,j)}\right) \end{aligned} \tag{4.1}$$

dove il pedice (i, j) identifica la posizione dell'elemento sulla matrice, mentre F rappresenta l'estensione spaziale del filtro.

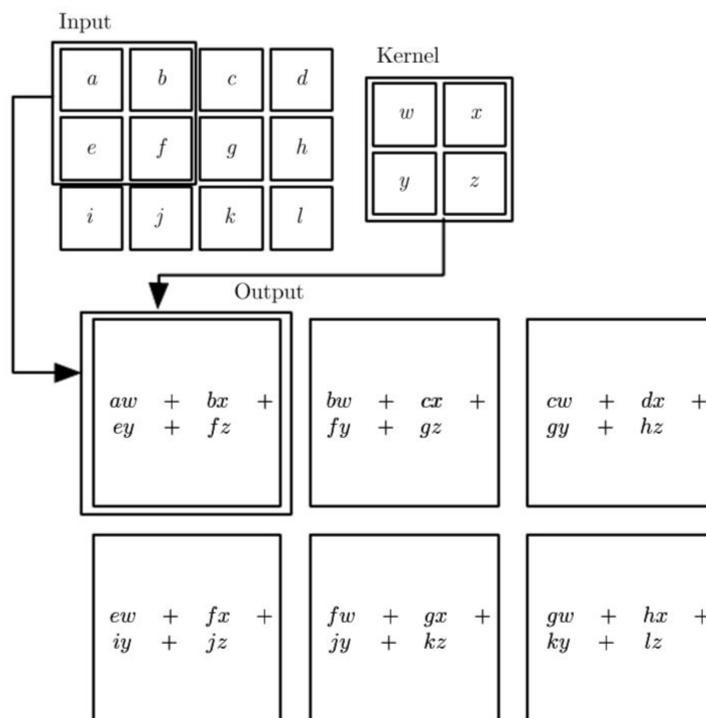


Figura 4.1: Operazione di convoluzione nel caso bidimensionale: un filtro di dimensione 2×2 viene moltiplicato elemento per elemento per una porzione dell'input di uguali dimensioni. L'output dell'operazione è costituito dalla somma di tali prodotti. L'operazione di convoluzione viene infine ripetuta spostando il filtro lungo le due dimensioni dell'input.

Nel caso in cui gli input siano rappresentati da volumi tridimensionali la procedura rimane la stessa, ma è importante notare che il filtro, pur mantendo una ridotta estensione spaziale (larghezza e altezza), viene esteso in profondità in misura uguale all'input. La figura 4.3 aiuta a comprendere meglio questo concetto: viene mostrato un esempio di convoluzione di due filtri (W0 e W1) su un input tridimensionale $(7 \times 7 \times 3)$. Per ragioni rappresentative i volumi sono stati scomposti in matrici bidimensionali, ma la profondità del filtro è la stessa di quella dell'input proprio perchè, mentre si muove lungo la superficie dell'input, deve andare ad operare lungo tutta la profondità. Questo significa che la proprietà di *sparse interaction* coinvolge solo le prime due dimensioni spaziali, ovvero larghezza e altezza, con la profondità che mantiene un approcio *fully connected.*

Formalmente, dovendo tenere conto anche della terza dimensione, la (4.1) diventa

$$\begin{aligned} z^{(2)}_{(i,j)} &= w^{(1)}_0 + \sum_{c=1}^{D} \sum_{a=1}^{F} \sum_{b=1}^{F} w^{(1)}_{(a,b,c)} x_{(i+a-1,j+b-1,c)} \\ a^{(2)}_{(i,j)} &= g^{(2)}\left(z^{(2)}_{(i,j)}\right) \end{aligned} \tag{4.2}$$

dove D rappresenta la profondità del volume di input (e di conseguenza del filtro).

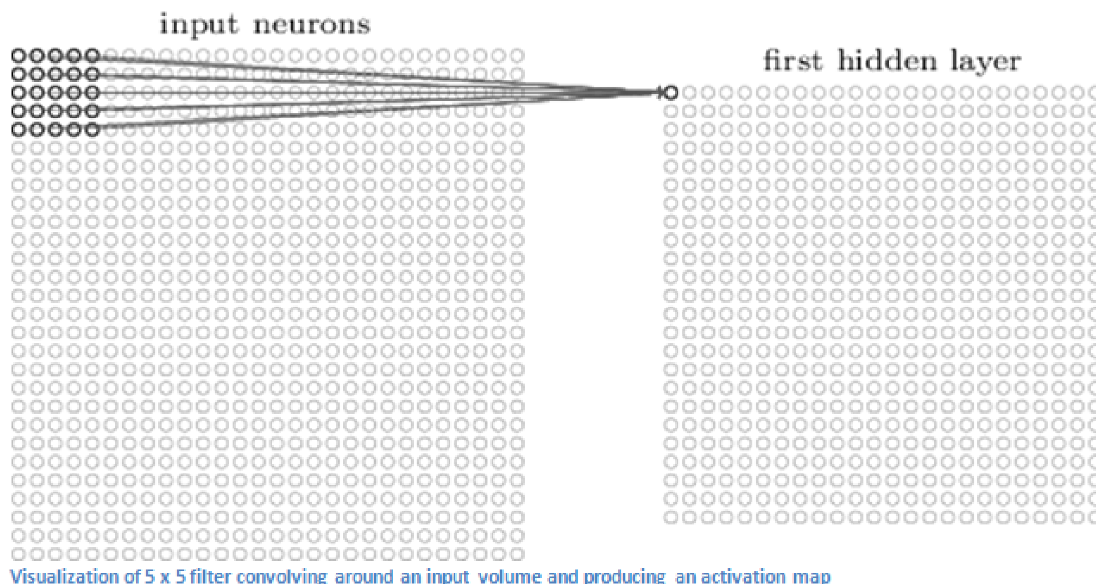


Figura 4.2: Operazione di convoluzione di un filtro di dimensioni 5×5 e relativa mappa di attivazioni prodotta.

É importante notare che la singola operazione di convoluzione produce sempre uno scalare, indipendentemente da quali siano le dimensioni del volume di input, sia esso bidimensionale o tridimensionale. Di conseguenza, una volta che il filtro viene fatto convolvere lungo tutta la superficie dell'input, si ottiene sempre una mappa di attivazioni bidimensionale.

4.2 Strati di una CNN

4.2.1 Convolutional Layer

Finora abbiamo sempre visto la convoluzione di un singolo filtro ma generalmente un *Convolutional Layer* è formato da un set di filtri molto più numeroso, diciamo N_F, tutti con la medesima estensione spaziale. Durante lo stage *feed-forward* ogni filtro viene fatto convolvere lungo la larghezza e l'altezza del volume di input, e pertanto vengono prodotte N_F mappe di attivazioni bidimensionali, le quali forniscono ognuna la risposta del relativo filtro in ogni posizione spaziale. La loro concatenazione lungo la terza dimensione produce l'output del *Convolutional Layer*.

Tenendo conto anche di quest'ultima considerazione, si può aggiornare la (4.2) introducendo l'indice relativo al filtro (f):

$$
\begin{aligned}
z^{(2)}_{f,(i,j)} &= w^{(1)}_0 + \sum_{c=1}^{D} \sum_{a=1}^{F} \sum_{b=1}^{F} w^{(1)}_{f,(a,b,c)} x_{(i+a-1,j+b-1,c)} \\
a^{(2)}_{f,(i,j)} &= g^{(2)} \left(z^{(2)}_{f,(i,j)} \right)
\end{aligned}
\tag{4.3}
$$

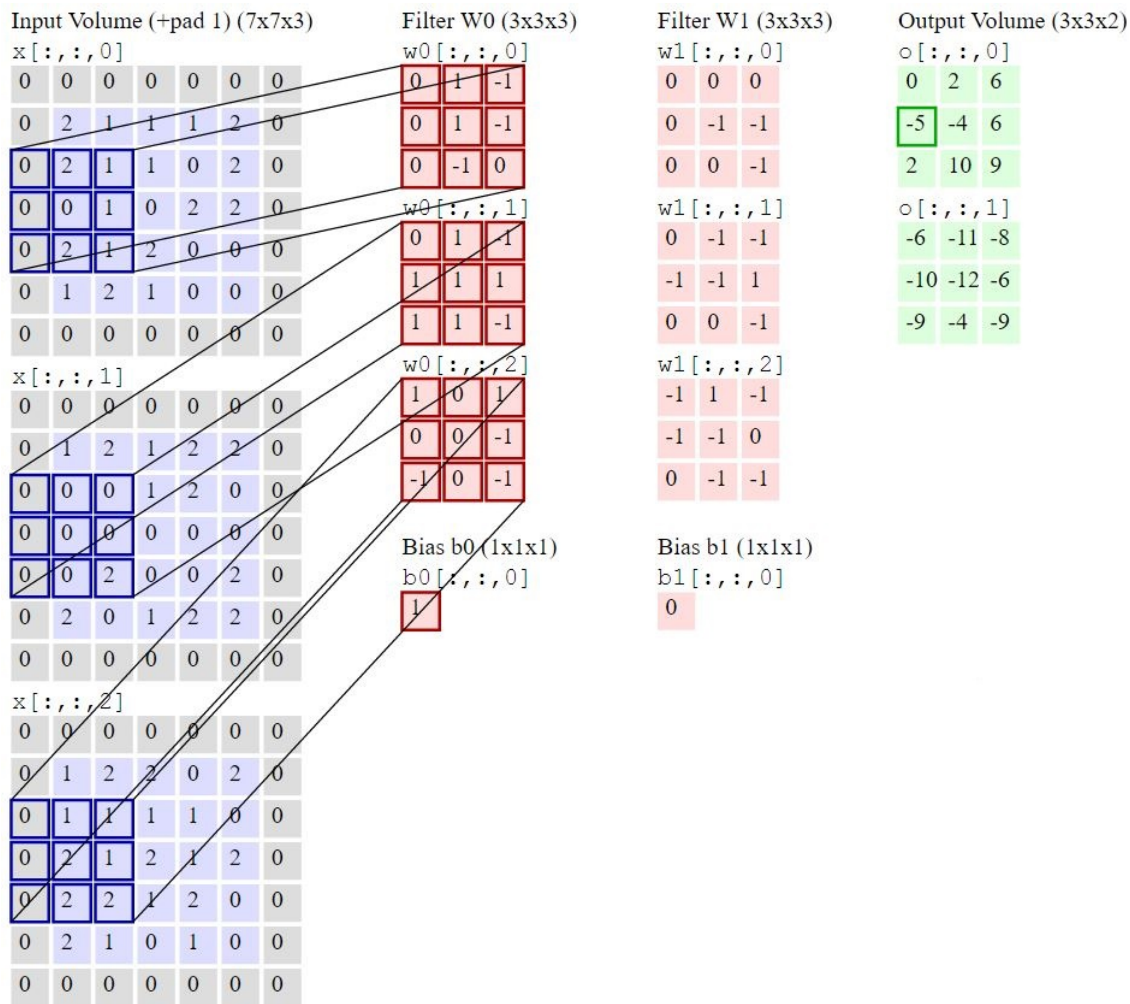


Figura 4.3: Operazione di convoluzione di due differenti filtri ($W0$ e $W1$) di dimensione $3 \times 3 \times 3$ su un volume di input $7 \times 7 \times 3$.

É facile notare che gli indici dei pesi che definiscono il singolo neruone non dipendono da (i, j), ovvero dalla sua posizione nella mappa di attivazione, ma solamente dal filtro generatore f. In altre parole, questo significa che i neuroni appartenenti alla stessa mappa di attivazione condividono sempre lo stesso set di pesi, ossia quelli del filtro che li ha generati. Questo definisce la seconda proprietà fondamentale delle Convolutional Neural Networks, ossia il ***weight sharing***.

Output dello strato

Disposizione finale e numero di neuroni che compongono il volume di ouput del *Convolutional layer* sono controllati da tre iperparametri: profondità, *stride*, e *zero-padding*, finora omessi per rendere le spiegazioni più semplici.

- **Profondità**: da non confondere con profondità della rete, corrisponde al numero di filtri N_F che compongono lo strato, ognuno in cerca di caratteristiche

differenti nel volume di input. Il set di neuroni (appartanenti a filtri diversi) connessi alla stessa regione di input prende invece il nome di *fibra*, o *colonna profondità*.

- **Stride**: specifica il numero di pixel di cui si vuole traslare il filtro ad ogni spostamento. Quando lo *stride* è pari ad uno significa che stiamo muovendo il filtro un pixel alla volta, e di conseguenza viene scannerizzata ogni possibile posizione dell'input. Valori più alti muovono il filtro con salti maggiori, e pertanto viene generato un output di dimensioni minori.

- **Zero-padding**: a volte può risultare conveniente aggiungere un bordo di zeri al volume di input, in modo così da controllare le dimensioni dell'output ed evitare incongruenze durante le operazioni. Lo spessore di questo bordo è determinato dall'iperparametro di *zero-padding*, ed è spesso utilizzato per far combaciare la dimensione dell'input con quella dell'output.

Larghezza e altezza del volume di output (O) possono essere calcolate come funzione della relativa dimensione nel volume di input (I), del *campo recettivo* del neurone (F), dello *stride* (S) applicato allo spostamento dei filtri, e della quantità di *zero-padding* (P) utilizzata per i bordi:

$$O = \frac{(I - F + 2P)}{S} + 1 \tag{4.4}$$

La profondità coincide invece con il numero di filtri N_F ultilizzati all'interno dello strato. Pertanto, dato in input un volume di dimensioni W_1 x H_1 x D_1 il *convolutional layer* produrrà un volume di output W_2 x H_2 x D_2 dove

$$\begin{aligned} W_2 &= \frac{(W_1 - F + 2P)}{S} + 1 \\ H_2 &= \frac{(H_1 - F + 2P)}{S} + 1 \\ D_2 &= N_F \end{aligned} \tag{4.5}$$

Chiaramente, i valori di stride e zero-padding devono essere scelti in modo tale che la (4.4) restituisca un valore intero. Ad esempio, l'architettura di Krizhevsky *et al.* (2012) accetta in input immagini di dimensione $[227 \times 227 \times 3]$ e utilizza nel suo primo *convolutional layer* 96 filtri di dimensione 11, *stride* pari a 4 e nessun *zero-padding*. Dal momento che $\frac{(227-11)}{4} + 1 = 55$ il volume finale dell'output del primo strato avrà dimensione $[55 \times 55 \times 96]$. Ognuno dei $55 \times 55 \times 96$ neuroni di questo volume è connesso ad una regione di dimensione $[11 \times 11 \times 3]$ del volume di input, e tutti i 96 neuroni appartenenti alla stessa *colonna profondità* sono connessi alla stessa regione $[11 \times 11 \times 3]$, ma ovviamente attraverso set diverso di pesi.

Weight Sharing & Invariance to Translation

Nell'esempio sopra riportato, ognuno dei $55 \times 55 \times 96 = 290\,400$ neuroni del primo *convolutional layer* possiede $11 \times 11 \times 3 = 363$ pesi, più uno relativo alla distorsione. In un'architettura come quella delle reti neurali classiche che non prevede la condivisione dei pesi questo comporterebbe un numero totale di parametri pari a $290\,400 \times 364 = 105\,705\,600$, solamente per il primo strato. Tale quantità, chiaramente intrattabile nella realtà, viene drasticamente ridotta dalla proprietà di *weight sharing* delle CNN, la quale si fonda su una ragionevole assunzione: se la rilevazione di una caratteristica in una determinata posizione spaziale risulta utile, lo sarà anche in differenti posizioni spaziali. Si assume cioè una struttura dell'immagine invarante rispetto a traslazioni. Tornando all'esempio, la proprietà di *weight sharing* comporta una diminuzione dei parametri per il primo strato fino a $(11 \times 11 \times 3) \times 96 + 96 = 34\,944$, rispetto ai precedenti 105 milioni, ovvero circa tremila volte meno.

Figura 4.4: Set di 96 filtri $11 \times 11 \times 3$ appresi dall'architettura di Krizhevsky *et al.* (2012) in un problema di classificazione di immagini. L'assunzione di *weight sharing* è ragionevole dal momento che individuare una linea o uno spigolo è importante in qualsiasi posizione dell'immagine, e di conseguenza non c'è la necessità di imparare ad localizzare la stessa caratteristica in tutte le possibili zone.

E' importante notare che in determinati contesti l'assunzione di *weight sharing* perde di senso. Un caso particolare è quando le immagini in input presentano una struttura specifica e centrata, e ci si aspetta di apprendere caratteristiche differenti in una determinata area dell'immagine piuttosto che in un'altra. Ad esempio, in un problema di riconoscimento facciale dove i volti sono stati ritagliati e centrati, risulterà più efficace apprendere nella parte superiore dell'immagine caratteristiche relative ad occhi o capelli, mentre nella parte inferiore quelle specifiche di bocca e naso.

4.2.2 Pooling Layer

Nell'architettura di una CNN è pratica comune inserire fra due o più *convolutional layers* uno strato di *Pooling*, la cui funzione è quella di ridurre progressivamente la dimensione spaziale degli input (larghezza e altezza), in modo da diminuire numero di parametri e carico computazionale, e di conseguenza controllare anche il sovra-dattamento. Il *Pooling Layer* opera indipendentemente su ogni mappa di attivazioni applicando un filtro di dimensione $F \times F$ che esegue una determinata operazione deterministica (tipicamente il massimo o la media), e pertanto non comporta la presenza di pesi. Anche qui, il calcolo del volume finale dipende, oltre che dalle dimensioni W_1 x H_1 x D_1 di input, dai due iperparametri richiesti, ossia stride (S) ed estensione spaziale del filtro (F):

$$
\begin{aligned}
W_2 &= \frac{(W_1 - F)}{S} + 1 \\
H_2 &= \frac{(H_1 - F)}{S} + 1 \\
D_2 &= D_1
\end{aligned}
\tag{4.6}
$$

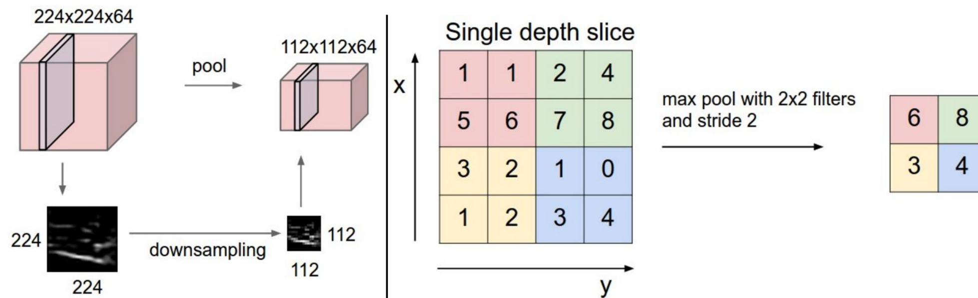


Figura 4.5: Esempio di max-pooling: il filtro opera indipendentemente su ogni *features-map* e di conseguenza la profondità del volume rimane inalterata. Larghezza e altezza vengono ridimensioante invece di un fattore $F = 2$ (di conseguenza viene eliminato il 75% dei pesi).

4.2.3 Fully-Connected Layer

Il *Fully-Connected Layer* è esattamente uguale ad un qualsiasi strato nascosto che compone le tradizionali reti neurali ed opera sul volume di output vettorizzato dello strato che lo precede. L'architettura *fully-connected* implica il rilassamento dell'assunzione di *weight sharing*: la funzione principale di tali strati, inseriti solo per ultimi a completare la struttura delle rete, è infatti quella di eseguire una sorta di raggruppamento delle informazioni ottenute negli strati precedenti, esprimendole attraverso un numero (l'attivazione neuronale) che servirà nei successivi calcoli per la classificazione finale. Intuitivamente, l'idea è quella che la rete apprenda filtri che

si attivino alla visione di determinati tipi di caratteristiche (*features*) come ad esempio angoli, linee o blocchi di colore nello strato iniziale (*features* di basso livello), oppure combinazioni via via sempre più complesse negli strati superiori (*features* di alto livello).

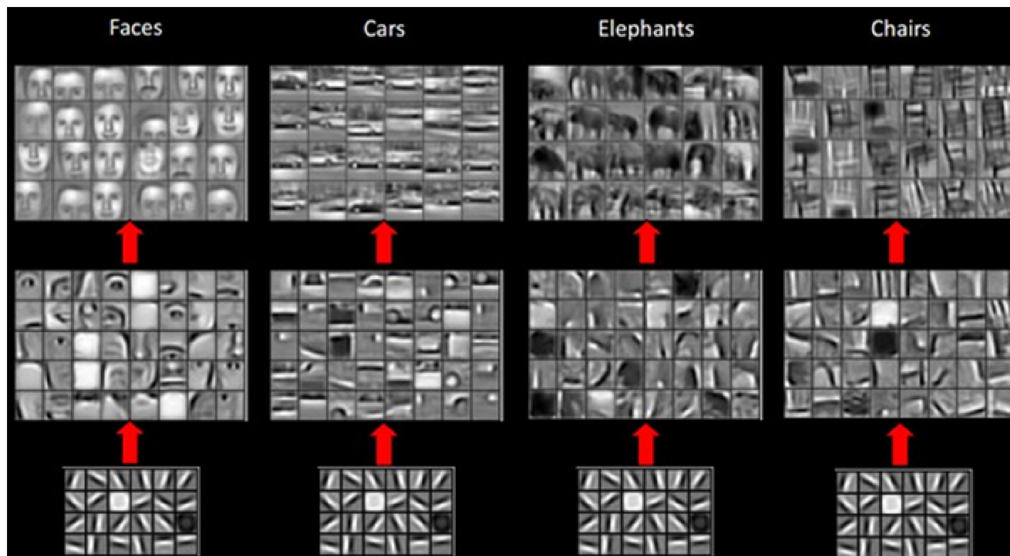


Figura 4.6: Esempio di apprendimento delle *features* su un dataset di oggetti misti.

4.3 Architettura generale della rete

Finora abbiamo visto i singoli strati che possono essere impiegati nell'architettura di una *Convolutional Neural Network*. Come per le reti neurali tradizionali, anche qui non esiste una linea guida per la scelta degli iperparametri, ne tantomeno per la sequenza da adottare nella scelta degli strati. Ci sono però alcune considerazioni ragionevoli che si possono fare per cercare quantomeno di muoversi lungo la giusta direzione.

La dimensione dei filtri dovrebbe, almeno in linea teorica, essere motivata dalla correlazione spaziale dell'input che un particolare strato riceve, mentre il numero di flitri utilizzati (quindi di *feature maps* generate) si riflette sulla capacità della rete di cogliere rappresentazioni gerarchiche: questo significa che gli strati finali necessitano di un numero di filtri maggiore di quello destinato agli strati iniziali, altrimenti si limitano le possibilità di combinare *features* di basso livello per rappresentare *features* di alto livello, che sono di fatto quelle più vicine alla classificazione finale. Altro fattore che incide sulla visione d'insieme della rete è invece la profondità intesa come numero di strati. Una rete con un singolo *convolutional layer* composto da filtri di dimensione 5×5 applicata ad immagini 500×500 non sarà mai in grado di offrire buone performance, neanche aumentando a dismisura il numero di filtri,

perchè verranno sempre colte solo le microstrutture dell'immagine. Aggiungere un secondo o un terzo *convolutional layer* non basta: è buona norma garantire un numero di strati tale da accumulare sufficiente campo visivo, in modo che il campo recettivo effettivo dei neuroni che compongo l'ultimo strato sia in grado di coprire l'intera estensione spaziale dei dati in input alla rete. É importante notare che ci si può muovere in questa direzione anche attraverso l'applicazione di operazioni che comprimono l'informazione spaziale in input allo strato successivo, come ad esempio le operazioni di *pooling* o l'aumento dello *stride* nelle operazioni di convoluzione. Queste soluzioni inoltre non comportano un incremento del numero di parametri, ed è il motivo per il quale spesso vengono inseriti strati di *pooling* fra due o più *convolutional layers* (si ricordi però che il prezzo da pagare è una perdita a livello di informazioni).

4.4 Il modulo *Inception*

Un dubbio ricorrente nella scelta degli iperparametri di una *Convolutional Neural Network* è quello relativo alla dimensione dei filtri da adottare per ogni strato. Intuitivamente uno potrebbe pensare che filtri più grandi siano adatti a cogliere *features* di scala maggiore e viceversa, ma l'esperienza insegna che in realtà la questione è molto più complessa.
Szegedy *et al.* (2015) hanno proposto una soluzione originale a questo problema: usare contemporaneamente filtri di dimensione diversa, e lasciare che sia la rete a decidere. Questa idea ha portato alla definizione del modulo *inception*, il quale, come mostrato in figura 4.7, prevede di eseguire in parallelo sullo stesso strato di input operazioni di convoluzione con filtri di dimensione 1×1, 3×3, 5×5, e di concatenare in output le mappe di attivazione ottenute.

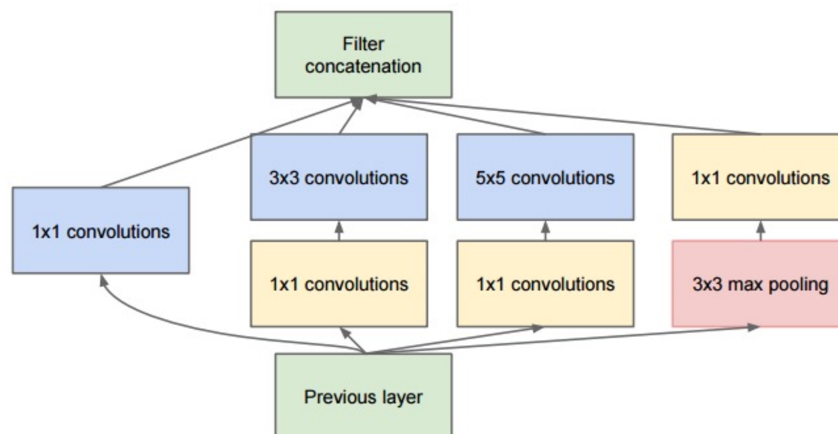


Figura 4.7: Architettura del modulo *inception*.

L'operazione di convoluzione 1×1 è la parte che più desta curiosità: questa estensione spaziale apparentemente priva di senso costituisce in realtà una forma intelligente di riduzione della dimensionalità. Consideriamo infatti il caso in cui un *convolutional layer* produca in output un volume di attivazioni di dimensioni $W \times H \times D$ (dove W, H, e D indicano rispettivamente larghezza altezza e profondità), e ipotizziamo che questo volume sia dato in input ad un *convolutional layer* composto da N_F filtri di dimensione 1×1: il volume di output che viene prodotto avrà dimensioni $W \times H \times N_F$, ovvero le stesse dimensioni spaziali (larghezza e altezza) del volume di input, ma profondità uguale ad N_F. Questo significa che possiamo utilizzare un *convolutional layer* con filtri 1×1 per modificare a piacere la profondità di un determinato volume di attivazioni, lasciando intatte le altre due dimensioni spaziali. In particolare, l'utilizzo di un solo filtro comporta la compressione del volume di input in una matrice. Questa è la ragione per la quale nel modulo *inception* viene inserita un'operazione di convoluzione 1×1 prima delle convoluzioni 3×3 e 5×5: se vogliamo espandere la rete in larghezza senza comportare un'esplosione del carico computazionale allora è necessario applicare una forma di riduzione della dimensionalità.

In realtà, la convoluzione 1×1 non nasce direttamente con lo scopo esclusivo di applicare una riduzione della dimensionalità, ma si basa su un determinato *background* teorico. Szegedy *et al.* (2015) sono infatti arrivati alla definizione di questa operazione ispirati dai risultati ottenuti da Arora *et al.* (2014): secondo quest'ultimi, se la distribuzione di probabilità di un dataset è rappresentabile da una rete multistrato sparsa, allora l'architettura ottimale della rete dovrebbe prevedere una costruzione strato per strato, analizzando le correlazioni delle attivazioni dell'ultimo strato e raggruppando in cluster i neuroni maggiormente correlati. Questi cluster dovrebbero quindi andare a formare le singole unità dello strato successivo. Intuitivamente, in una *Convolutional Neural Network* risulta facile pensare che tali gruppi di unità correlate siano rappresentati dai neuroni appartenenti alla stessa fibra (o *depth column*), dal momento che questi sono tutti collegati alla stessa regione dello strato precedente. Di conseguenza, l'operazione di raggruppamento può essere rappresentata nell'architettura di una CNN tramite la definizione di un *convolutional layer* con filtri di dimensione 1×1.

La presenza di una quarta operazione in parallelo, identificata dallo strato di *max pooling*, non è giustificata dagli autori del modulo *inception* tramite una valida motivazione teorica, ma piuttosto da ragioni empiriche, dal momento che, a detta degli stessi, storicamente gli strati di *max pooling* aiutano a migliorare le *performance* della rete. In ogni caso è bene sottolineare che la riduzione della dimensionalità operata dall'operazione di pooling opera in direzione opposta a quella offerta dalla convoluzione di filtri 1×1: la prima agisce sulle due principali dimensioni spaziali riducendo larghezza e altezza, mentre la seconda regola la profondità del volume di input. Inoltre, ricordiamo che l'operazione di *pooling* applica una trasformazione deterministica, mentre l'operazione di convoluzione è una procedura parametrizzata che restituisce quella combinazione lineare degli input che meglio riassume le

informazioni di interesse.

Tornando al modulo *inception*, il vantaggio di adottare questa architettura risiede nel fatto che è possibile estendere in maniera importante la larghezza della rete senza avere grosse ripercussioni sulla complessità computazionale. Inoltre, l'utilizzo contemporaneo di filtri di dimensione diversa si trova perfettamente in accordo con l'intuizione che l'informazione visiva dovrebbe essere processata a varie grandezze e successivamente aggregata, in modo che lo strato successivo sia in grado di estrarre simultaneamente *features* di dimensione diversa.

4.5 Data Augmentation

Il metodo migliore di stimare un modello in grado di offrire un buona performance in termini di generalizzazione è senza dubbio quello di allenarlo su un dataset il più ampio possibile; purtroppo però nella pratica la quantità di dati disponibili è sempre limitata. Un modo per aggirare questo problema è quello di generare delle osservazioni "false" da aggiungere al *training set*, a partire da versioni leggermente modificate dei dati originali. Questa tecnica è particolarmente efficace in problemi di classificazione di immagini, in quanto quest'ultime sono influenzate dalla presenza di una vasta gamma di fattori di variazione, molti dei quali possono essere facilmente simulati. Ad esempio, traslare di pochi pixel l'immagine in una qualsiasi direzione può incrementare le capacità di generalizzazione di una *Convolutional Neural Network*, anche se le proprietà di quest'ultima offrono già invarianza rispetto a traslazioni. Altre trasformazioni utili sono la riflessione e la rotazione, ma sono da applicare con più prudenza in quanto è necessario prima assicurarsi che non influiscano sulla corretta classificazione delle osservazioni: in un dataset composto da immagini che ritraggono numeri ad esempio, un'operazione di rotazione di 180° comporterebbe la trasformazione di un "6" in un "9", o viceversa.

Capitolo 5

Dataset

Trovare una database sul quale stimare i modelli proposti si è rivelato un compito abbastanza complesso, in quanto a causa della natura dei contenuti è difficile che chi si occupa di queste analisi renda pubbliche le immagini su cui lavora, a maggior ragione quando il loro collezionamento richiede un dispendio non trascurabile in termini di tempo e risorse.

5.1 NPDI Database

L'unico dataset reperibile in rete è fornito su richiesta dal Digital Image Processing Group (NPDI) dell'Università federale brasiliana di Minas Gerais, che ne autorizza l'accesso solamente ad istituzioni legali a fini di ricerca senza scopo di lucro. Il database contiene 16727 immagini campionate casualmente da video e divise in tre categorie a seconda del filmato dal quale sono state prelevate:

- 6785 immagini classificate come non pornografiche semplici, campionate da video che ritraggono scene di tutti i giorni, dove le persone che compaiono sono sempre vestite.
- 3555 immagini prelevate da video selezionati da query come "nuotatori", "spiaggia", "wrestling", "neonati", etc. Queste immagini sono etichettate come non pornografiche difficili in quanto contengono molte parti del corpo scoperte, senza però mostrare contenuti sessualmente espliciti.
- 6387 immagini campionate da video pornografici di vario genere e con attori di diversa etnia.

Maggiori informazioni relative ai video dai quali le immagini sono state campionate sono riassunte nelle tabelle 5.1 e 5.2.

Categoria	Video	Ore	Media campionamenti per video
Non pornografiche semplici	200	11.5	33.8
Non pornografiche difficili	200	8.5	17.5
Pornografiche	400	57	15.6
Totale	800	77	20.6

Tabella 5.1: Statistiche dei video dai quali sono state campionate le immagini.

Etnia	% nei video
Asiatici	16 %
Neri	14 %
Bianchi	46 %
Multietnici	24 %

Tabella 5.2: Etnia degli attori nei video pornografici

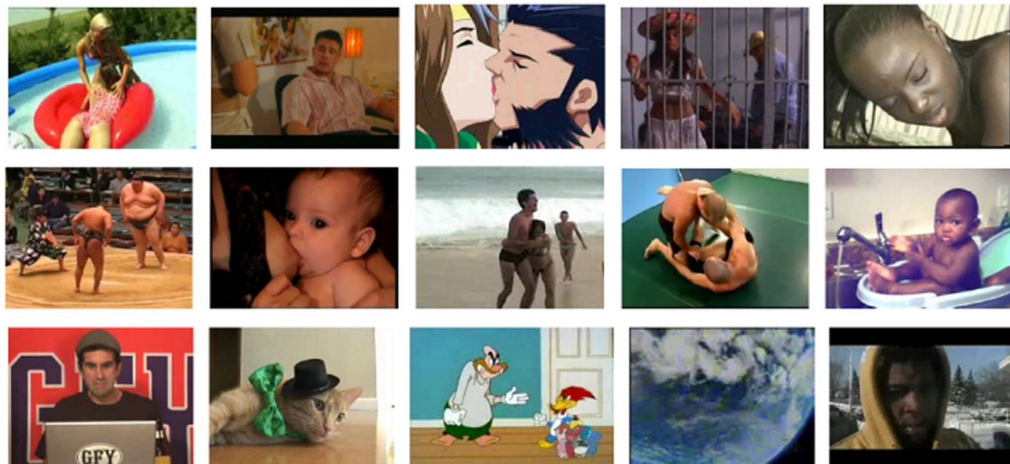


Figura 5.1: Alcune immagini del database, divise per categoria in base alla riga.

Il punto forte del dataset è la sua composizione molto variegata, in particolare per quanto riguarda la categoria di interesse, che può far pensare di ottenere in fase di stima dei risultati ben generalizzati. D'altra parte però, ci sono svariati difetti su cui è bene soffermarsi. Il primo è che le immagini sono in realtà frame di video che sono stati pesantemente compressi per essere caricati in rete: il risultato è una qualità del file molto scadente e la presenza di artefatti causati dagli aggressivi algoritmi di compressione. Ad ogni modo questo non rappresenta un grande problema in quanto si tratta solo di una riduzione dell'informazione che la singola immagine è in grado di fornire, e dell'introduzione di rumore (gli artefatti) che nella più ottimistica delle ipotesi può anche agire come regolarizzatore. Le problematiche più grandi derivano invece dal fatto che le 16727 immagini sono state estratte da soli 800 video, con una media di 20 campionamenti per ognuno. Questo significa che le immagini appartenenti allo stesso video risultano essere molto correlate fra loro: condividono

infatti spesso le stesse condizioni di luminosità, sfondo, colori predominanti, tutte variabili che messe assieme possono tradursi quasi in un fattore *leaker* che ci porta ad avere una buona accuratezza in fase di testing, ma che sta però ingannando il modello, distogliendo l'attenzione dalle vere caratteristiche discriminanti del problema. Questo problema per giunta affligge maggiormente la categoria d'interesse, dal momento che i video pornografici sono anche quelli più "statici", con lunghe inquadrature che rappresentano sempre la stessa scena, e con la conseguenza che risulta più probabile campionare immagini simili. L'ultimo difetto rilevante è invece la presenza di un numero significativo di falsi positivi, all'incirca il 5% in relazione all'intero dataset. Questi sono il prodotto di un campionamento effettuato in maniera del tutto casuale, con il risultato che sono sono presenti immagini catalogate come pornografiche che invece rappresentano titoli di testa, titoli di coda, o scene iniziali in cui non c'è presenza di nudo in quanto gli attori sono ancora vestiti.
Tutte queste problematiche rappresentano un ostacolo per lo sviluppo di un modello in grado di focalizzarsi sul fulcro del problema, ed hanno pertanto spinto alla creazione di un database che costituisse una solida base di partenza.

5.2 Web Crawling

Procedere attraverso lo scaricamento manuale e individuale di immagini dal web rappresenta un'idea malsana oltre che dispendiosa in termini di tempo, pertanto si è deciso di ricorrere a soluzioni più efficienti: i Web Crawlers. Un Web Crawler (conosciuto anche come Web Spyder) è un programma automatizzato, uno script che scannerizza in maniera metodica le pagine web per creare un indice dei contenuti per il quale è stato configurato. Esistono sono svariati utilizzi che si possono fare dei Web Crawlers, ma il principale rimane quello di collezionare dati da internet in grosse quantità, motivo per il quale la pratica del Web Crawling viene spesso affiancata alle attività di Data Mining.
In questa tesi è stato utilizzato il Web Crawling con lo scopo di scaricare in blocco immagini a partire da query google in maniera ricorsiva, attraverso la creazione di un file di testo contenente la lista query da eseguire e i domini da esplorare. Per ovvie ragioni le query non saranno qui elencate, ma si è cercato di comprendere il maggior numero categorie e soggetti, sulla falsariga del database NPDI, in modo da ottenere un dataset il più completo e variegato possibile. Per quanto riguarda la parte "safe" del database si è deciso invece di utilizzare due grandi dataset *open source* disponibili in rete: il PIPA (People In Photo Album) e il MPII Human Pose dataset. Il primo contiene circa 60'000 foto prelevate dagli album personali di 2000 persone caricati pubblicamente su Flickr (figura 5.2), mentre il secondo contiene 25'000 immagini raffiguranti oltre 40'000 persone impiegate in 410 diverse attività (figura 5.3).

Figura 5.2: Esempio di immagini che compongono il dataset PIPA.

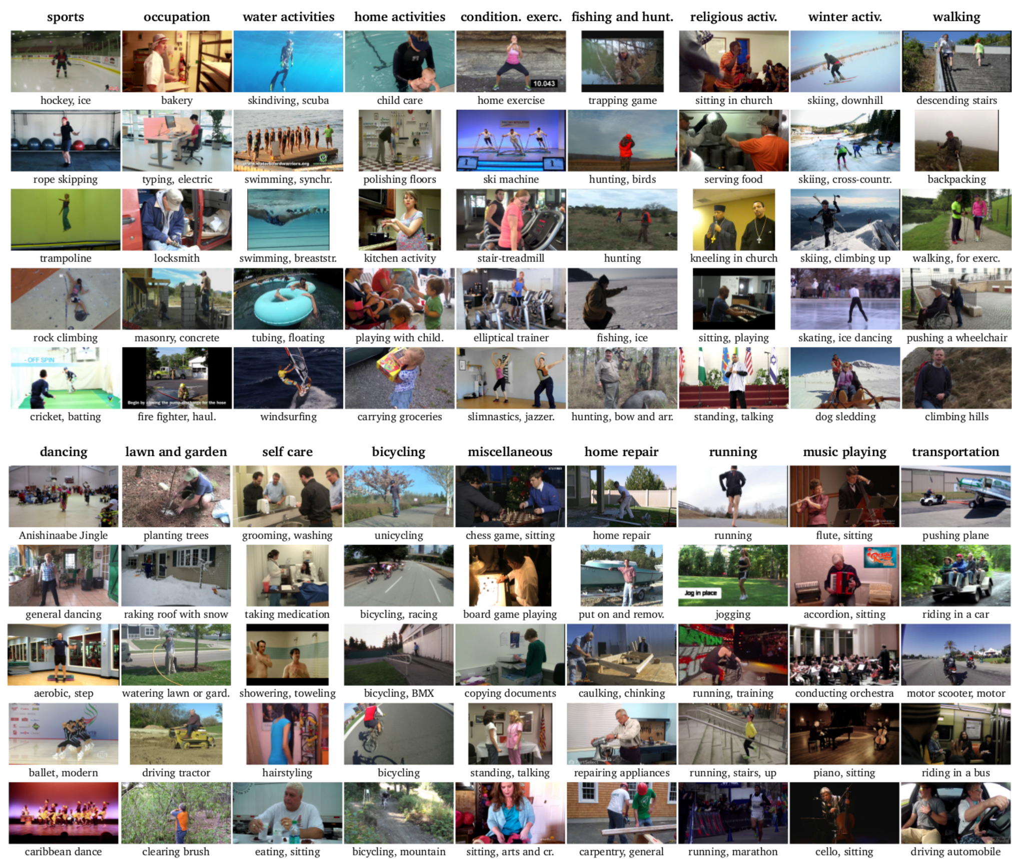


Figura 5.3: Alcune delle 410 attività rappresentate nel database MPII Human Pose dataset, divise per categoria (colonne).

In totale il database è composto da 130'000 immagini, di cui 65'000 a contenuti espliciti collezionate mediante Web Crawling, e 65'000 campionate casualmente dai dataset PIPA e MPII Human Pose.

Capitolo 6

Analisi e Modelli

6.1 Primi modelli sul dataset NPDI

Preprocessing

Innanzitutto è stata necessaria una fase di *preprocessing* dei dati, al fine di agevolare le successive analisi e rendere quest'ultime sostenibili dal punto di vista computazionale. In questo senso verranno fatte alcune piccole considerazioni tecniche. Il database NPDI è composto da immagini di dimensioni variabili in base al video di provenienza, entro un range che va dai 192×144 pixel per lato di quelle più piccole ai 720×480 di quelle più grandi. Salvate attraverso il famoso formato di compressione JPEG queste occupano su disco poco più di 300 MegaBytes, ma come sappiamo i modelli non possono lavorare su questo tipo di codifica: hanno bisogno delle matrici di dati grezze che compongono l'immagine, ossia dei singoli valori numerici che rappresentano il sub-pixel (ogni pixel è formato da tre numeri che identificano l'intensità dei colori RGB - Red Green Blue). Tenendo conto che la dimensione media delle immagini è di circa 480×360 pixel questo significa che le 16 727 immagini del dataset sono composte complessivamente da $480 \times 360 \times 3 \times 16\,727 = 8.7$ miliardi di numeri. Dal momento che in R i singoli numeri sono allocati in celle di memoria della dimensione di 8 bytes ognuna, questo significa che dovremmo disporre di circa 64 GigaByte di memoria RAM solo per poter caricare l'intero dataset, senza contare la fetta di memoria riservata al sistema operativo e quella necessaria al *software* per eseguire i calcoli.
Per queste ragioni, unite al fatto che i modelli necessitano di input di dimensioni fisse, le immagini sono state ridimensionate tutte alla risoluzione finale di 120×90, riducendo il carico di memoria a circa 4 GigaByte. Dal momento che le immagini originali presentano tutte proporzioni diverse sono state valutate tre modalità per il ridimensionamento (tabella 6.1): (a) mantenere l'*aspect-ratio*, ovvero le proporzioni originali, forzando l'immagine dentro la dimensione finale e aggiungendo delle bande nere orizzontali/verticali in modo da riempire le zone lasciate scoperte; (b) effettuare un ritaglio centrale 120×90 dell'immagine ridimensionata adattando il

Tabella 6.1: Diverse modalità valutate per il ridimensionamento delle immagini, a partire dalla versione originale (in alto).

lato più lungo, (c) forzare l'immagine dentro il riquadro finale perdendo l'apect-ratio originale, ossia comprimendo il lato che altrimenti sarebbe fuoriuscito dal frame.

Infine, si è deciso di optare per la modalità (c) alla luce di due considerazioni: è quella che comporta la minore perdita di informazione e al tempo stesso l'eventuale modifica dell'*aspect-ratio* dovrebbe rendere il modello più robusto a dilatazioni e distorsioni prospettiche.

6.1.1 Gradient Boosting

Il primo modello che verrà presentato è un *gradient boosting* stimato sulla matrice vettorizzata dei singoli sub-pixel, quindi su un totale di $120 \times 90 \times 3 = 32400$ variabili per osservazione. Questo numero è anche la causa principale dell'assenza in questa tesi di altri modelli di stima che sarebbe stato interessante valutare, come ad esempio le *Support Vector Machines* o l'analisi discriminante. Tali modelli si sono rivelati infatti inaccessibili per via della richiesta insostenibile in termini di tempi e costi computazionali, sproporzionata per l'hardware a disposizione. Non è stato possibile nemmeno operare sulle componenti principali, in quanto gli algoritmi implementati in R hanno saturato velocemente i 16 GigaByte di RAM disponibili, anche su *subset* ridotti del database.

Come anticipato nel capitolo primo, per la stima del modello *Gradient Boosting* ci si è serviti della libreria R *xgboost* di Chen e Guestrin (2016), e più nello specifico della funzione *xgb.train*. Questa prevede diversi parametri di regolazione che permettono un'efficace messa a punto del modello, fra i quali anche alcuni che gestiscono la possibilità di introdurre un termine di regolarizzazione sui pesi. I più importanti sono:

- *eta*: è il tasso di apprendimento ϵ (algoritmo 2.4, capitolo 2). Valori piccoli implicano una convergenza più lenta ma maggiore robustezza contro il sovradattamento.
- *gamma*: rappresenta la riduzione minima sulla funzione di perdita richiesta per effettuare uno *split* sull'albero.
- *max_depth*: profondità massima raggiungibile dal singolo albero.
- *subsample*: proporzione di osservazioni estratte casualmente ad ogni iterazione.
- *colsample_bytree*: proporzione di variabili estratte casualmente durante la costruzione dei singoli alberi.
- *num_parallel_tree*: numero di alberi da stimare in ogni iterazione (valori maggiori di uno permettono di costruire una foresta casuale).
- *lambda*: termine di regolarizzazione L2 (weight decay) sui pesi.

La scelta finale dei parametri è stata fatta attraverso una procedura di convalida incrociata ripetuta (con stesso seme per il generatore dei numeri pseudocasuali) su un set di combinazioni ragionevoli. In particolare, i valori che hanno minimizzato l'errore di classificazione sul test set sono stati i seguenti: $[\epsilon = 0.2, max_depth = 6, lambda = 0.000001]$ I risultati in tabella 6.2 mostrano le performance in termini di accuratezza ottenute su tre diverse composizioni del training/test set che differiscono per come è stato definito l'insieme dei campioni negativi (solo immagini non pornografiche facili, difficili, o entrambe).

Composizione training/test set	Accuratezza
Facile	87.21 %
Difficile	78.74 %
Misto	83.40 %

Tabella 6.2: Risultati ottenuti con un modello gradient boosting sui singoli pixel per diverse composizioni del dataset.

Eventuali termini di interazione non sono stati aggiunti perchè gli alberi che costituiscono il modello sono già portati a cogliere strutture di correlazione nei dati, e solo l'introduzione di interazioni binarie avrebbe comportato un carico di altri $p(p-1)/2$ parametri. Pertanto, come anticipato nel primo capitolo, si è deciso di portare informazioni aggiuntive al modello attraverso misure di sintesi discriminanti quali il numero totale di pixel identificati come pelle e la porzione appartenente al volto di quest'ultimi. Si sono svolte dunque altre operazioni di *preprocessing* attraverso l'utilizzo delle librerie di computer vision disponibili per il linguaggio python, ovvero *openCV* e *openCV2*: per quanto riguarda l'identificazione della pelle è stato utilizzato un algoritmo a soglie multiple (ognuna su un diverso spazio colore) che ha prodotto per ogni immagine una maschera binaria composta da pixel bianchi o

neri, dove il bianco identifica il pixel classificato come pelle (figura 6.2).
Per il conteggio dei pixel appartenenti alla faccia è stato prima necessario utilizzare un algoritmo di localizzazione dei volti (2.2.2): una volta ottenute le coordinate dei rettangoli contenenti i volti dell'immagine (figura 6.1) si sono ritagliate le corrispettive regioni sulla maschera binaria prodotta in precedenza e si è effettuato il conteggio totale dei pixel classificati come pelle.

Figura 6.1: Esempio di localizzazione del volto.

Figura 6.2: Maschera binaria dei pixel identificati come pelle (colore bianco).

La stima del modello è stata quindi ripetuta sul dataset aggiornato con l'aggiunta di 5 variabili esplicative:

1. conteggio pixel classificati come pelle
2. conteggio pixel classificati come pelle appartenenti ai volti
3. numero di volti identificati nell'immagine
4. percentuale di pixel classificati come pelle nell'intera immagine
5. rapporto fra le prime due variabili aggiunte

E' importante sottolineare che, per ottenere risultati più affidabili, le operazioni di *preprocessing* che hanno generato le nuove variabili sono state effettuate sulle immagini originali e non le loro versioni ridimensionate. Di conseguenza la quarta variabile aggiunta non è una combinazione lineare della prima come potrebbe essere naturale pensare.

I risultati ottenuti con la versione aggiornata del dataset sono visibili nella tabella 6.3, mentre la figura 6.4 riporta l'errore in funzione dell'iterazione.

Composizione training/test set	Accuratezza
Facile	87.21 %
Difficile	78.74 %
Misto	83.40 %

Tabella 6.3: Risultati ottenuti con un modello gradient boosting sui singoli pixel con l'aggiunta delle variabili relative alle informazioni sulla pelle.

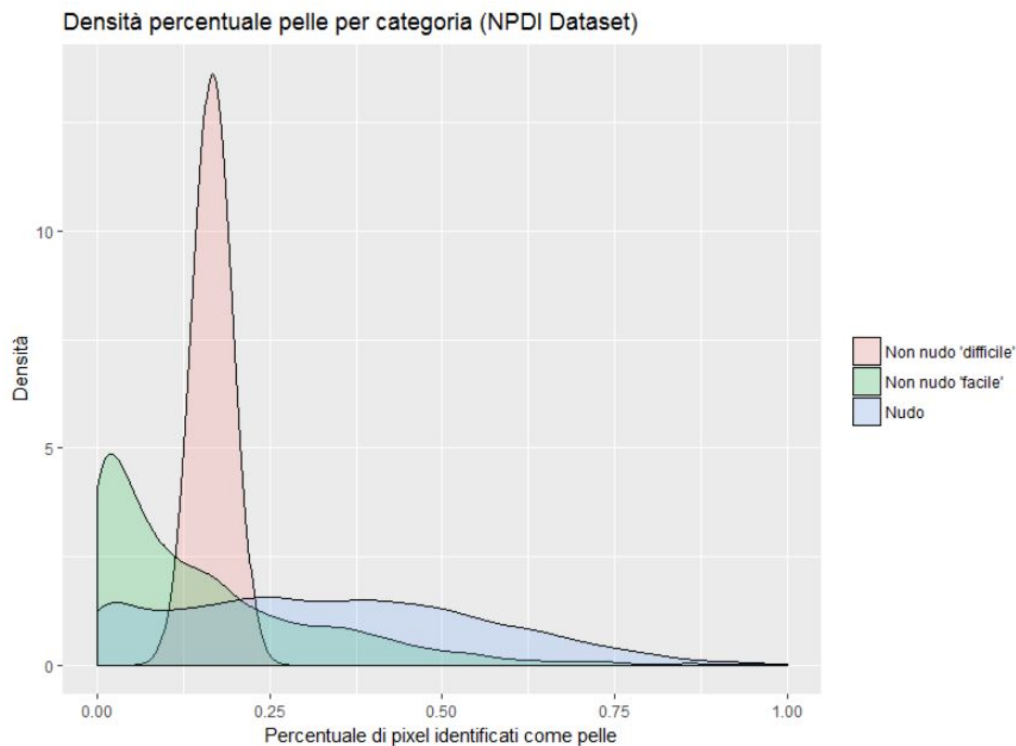


Figura 6.3: Curve di densità relative alla percentuale di pelle nelle immagini, divise in base alla categoria di appartenenza.

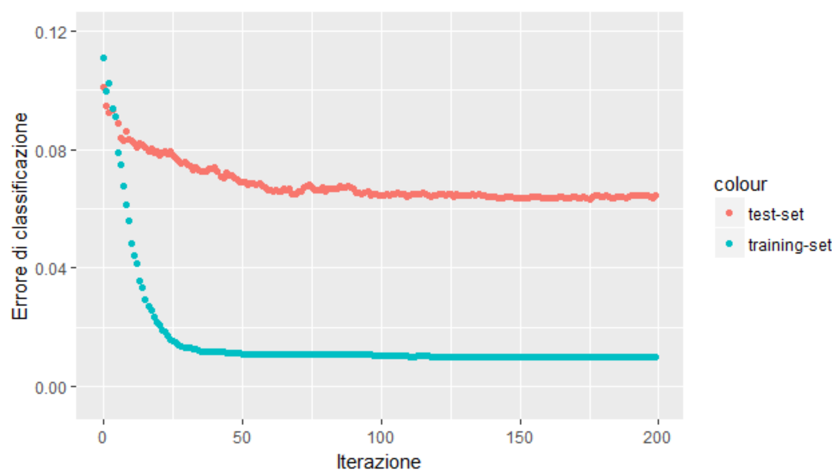


Figura 6.4: Errore di classificazione per training e test set, in funzione dell'iterazione. Il dataset utilizzato è quello intero, ovvero con composizione mista.

6.1.2 Convolutional Neural Networks

Per rendere la stima delle Convolutional Neural Networks sostenibile dal punto di vista computazionale è stata necessaria l'installazione di un *framework* che consentisse di svolgere le operazioni di calcolo sulla GPU piuttosto che sulla CPU. L'architet-

tura delle schede video combinata alla predisposizione al calcolo parallelo delle reti neurali permette infatti di accelerare di molto (all'incirca 10 volte) i tempi di stima passando, nel caso dei modelli più complessi, da settimane a giorni (in contesti ovviamente di *big data*). In particolare, la scelta del *framework* è ricaduta su *mxnet* (Chen *et al.*, 2015).

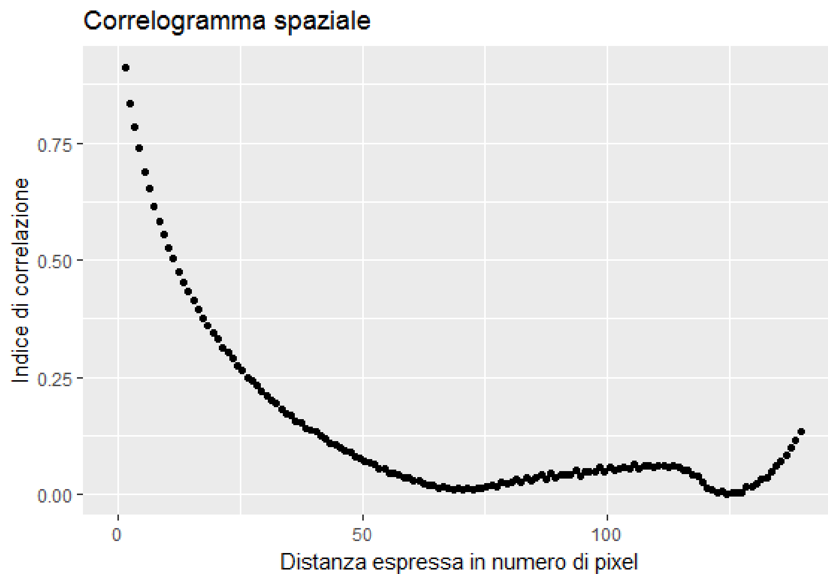


Figura 6.5: Correlogramma spaziale calcolato sulle immagini del dataset NPDI.

Per disegnare l'architettura della prima CNN si è voluto seguire almeno inizialmente un approccio teorico, andando ad analizzare la correlazione spaziale delle immagini per scegliere di conseguenza la dimensione dei filtri del primo *convolutional layer*. Il correlogramma spaziale (figura 6.5) evidenzia presenza di correlazione forte ($>$0.75) fino al lag 3, dove il lag indica la distanza fra pixel allineati. Di conseguenza si è optato per un'estensione spaziale dei filtri del primo strato pari a 7×7 (pixel centrale ± 3). Numero di filtri per strato e numero di strati della rete sono stati invece scelti seguendo un'approccio più sperimentale, secondo quanto già spiegato nel paragrafo (4.3). In particolare si è dimostrato essenziale l'inserimento a cascata di un numero sufficiente di *convolutional layers* e un'opportuna regolarizzazione negli strati finali della rete, che hanno portato alla definizione di una prima architettura così definita:

1. *Convolutional layer* #1: 64 filtri di dimensione 7×7, *stride* 1, *zero-padding* 0;

2. *Convolutional layer* #2: 128 filtri di dimensione 5×5, *stride* 1, *zero-padding* 0;

3. *Pooling layer* #1: filtro di dimensione 2×2, *stride* 2, operazione di massimo;

4. *Convolutional layer* #3: 192 filtri di dimensione 3×3, *stride* 1, *zero-padding* 0;

5. *Convolutional layer* #4: 256 filtri di dimensione 3×3, *stride* 1, *zero-padding* 0;

6. *Pooling layer* #2: filtro di dimensione 3×3, *stride* 3, operazione di massimo;

7. *Convolutional layer* #5: 328 filtri di dimensione 3×3, *stride* 1, *zero-padding* 0;

8. *Convolutional layer* #6: 384 filtri di dimensione 3×3, *stride* 1, *zero-padding* 0;

9. *Fully connected layer* #1: 256 unità;

10. *Dropout* #1: $\phi = 0.5$

11. *Fully connected layer* #2: 128 unità;

12. *Dropout* #2: $\phi = 0.5$

Per quanto riguarda le funzioni di attivazione i migliori risultati, sia in termini di velocità di convergenza che di accuratezza finale, sono stati raggiunti con la ReLU. La stima della rete è avvenuta invece tramite *Stochastic Gradient Descent* con dimensione dei *batch* calcolata in modo da ottimizzare i tempi di stima (si rimanda al capitolo 7 per maggiori approfondimenti), mentre i vari algoritmi di ottimizzazione della discesa del gradiente non hanno portato significativi miglioramenti. In particolare, un'attenta inizializzazione del *learning rate* seguita da una sua progressiva riduzione all'aumentare delle epoche, ha portato a risultati simili (e in alcuni casi migliori) a quelli ottenuti con Adam (equazione 3.24).
In questa configurazione la rete ha registrato un errore su test set tramite convalida incrociata pari all'1.92%, ovvero molto al di sotto del tasso di falsi positivi che caratterizza il dataset NPDI (circa il 5%), a significare che il modello ha imparato a classificare come immagini contenenti nudo anche i falsi positivi. Altre possibili cause di un'accuratezza che sembra essere fin troppo ottimistica sono da identificare invece nel problema già accennato della composizione a cluster di immagini correlate che compone il dataset NPDI (5.1). Per esprimere al meglio le potenzialità delle *Convolutional Neural Networks* e al tempo stesso costruire un classificatore in grado di cogliere le vere caratteristiche discriminanti si è deciso quindi di utilizzare un proprio dataset affidandosi alla pratica del *Web Crawling* (5.2).

6.2 Analisi sul "*Crawling* dataset"

Con il nuovo dataset si è deciso anche di cambiare approccio per quanto riguarda la fase di preprocessing, ridimensionando le immagini ad una risoluzione di 240×240.

L'*aspect-ratio* 1:1 è motivato da due ragioni. La prima risiede nella diversa natura del dataset: non si tratta più di frame campionati da video 4:3 o 16:9, ma di immagini scaricate dal web, che presentano mediamente un *aspect-ratio* quadrato. La seconda è che avere un input con lati uguali permette più flessibilità in fase di progettazione, in particolare per quanto riguarda la scelta di *stride*, *zero-padding*, e dimensione dei filtri, dal momento che ci sono meno vincoli da soddisfare (equazioni 4.5).
L'incremento generale delle dimensioni è invece permesso dal fatto che la discesa del gradiente SGD (3.3.2) rende il modello stimabile in maniera ricorsiva, un *batch* di dati per volta. Di conseguenza non è più necessario caricare l'intero dataset in memoria, ma solo il batch di osservazioni corrente. A livello computazionale, questo è reso possibile attraverso la definizione di un iteratore, il quale verrà approfondito nel capitolo successivo. Ovviamente l'aumento delle informazioni in input (maggior numero di immagini e maggiore risoluzione) ha comportato l'impossibilità di continuare ad utilizzare il *gradient boosting*, non potendo quest'ultimo essere stimato in maniera ricorsiva.
Infine, questo tipo di *preprocessing* è stato ripetuto anche sul dataset NPDI, in modo da poter essere utilizzato come insieme di convalida.

6.2.1 Data Augmentation

Nonostante siano state prodotte in fase di *preprocessing* immagini di dimensione 240×240 si è deciso di costruire le architetture delle *Convolutional Neural Networks* per ricevere in input immagini di dimensione 224×224. Questo allo scopo di garantire un margine di ulteriore preprocessing per le operazioni di *data augmentation* (4.5). Le reti successive sono state infatti stimate su trasformazioni delle immagini originali: ad ogni epoca e per ogni immagine è stato effettuato un ritaglio casuale 224×224 sulle dimensioni di partenza (240×240) e il risultato è stato specchiato con probabilità 0.5. In questo modo si è ottenuto un potenziale aumento dei dati di un fattore $(240 - 224) \times (240 - 224) \times 2 = 512$, allo scopo di limitare il sovradattamento. Ovviamente questa seconda fase di *preprocessing* si è svolta "*online*", ovvero solo all'occorrenza durante la fase di training, altrimenti avrebbe comportato il salvataggio di $130\,000 \times 512$ immagini (oltre 66 milioni).

6.2.2 Più strati, filtri più piccoli

Le reti presentate in questa sezione sono modifiche di architetture già esistenti in letteratura, a cui ci si è ispirati per ragioni di tempo: sarebbe stato infatti troppo dispendioso continuare con un approccio sperimentale, dal momento che la stima di modelli così complessi su dataset di dimensioni altrettanto importanti richiede almeno 24h sulle configurazioni hardware più recenti.
La prima rete prende spunto dall'architettura VGG di Simonyan e Zisserman (2014), secondo i quali le capacità di rappresentazione della rete aumentano se si limita il

#	Tipo strato	Numero filtri	Dimensione filtri	Stride	Zero-padding
1	Convolutional	64	3×3	1	1
2	Pooling	1	2×2	2	
3	Convolutional	128	3×3	1	1
4	Pooling	1	2×2	2	
5	Convolutional	256	3×3	1	1
6	Convolutional	256	3×3	1	1
7	Pooling	1	2×2	2	
8	Convolutional	512	3×3	1	1
9	Convolutional	512	3×3	1	1
10	Pooling	1	2×2	2	
11	Convolutional	512	3×3	1	1
12	Convolutional	512	3×3	1	1
13	Pooling	1	2×2	2	
14	Fully Connected	2048 neuroni			
15	Dropout	$\phi = 0.5$			
16	Fully Connected	2048 neuroni			
17	Dropout	$\phi = 0.5$			

Tabella 6.4: Architettura disegnata sulla falsariga di quella proposta da Simonyan e Zisserman (2014) e ridimensionata all'hardware a disposizione.

campo recettivo dei neuroni ad una finestra 3×3 e si aumenta la profondità attraverso un numero consistente di *convolutional layers* in successione. In particolare, è interessante notare che il campo recettivo effettivo di tre successivi *convolutional layer* con filtri 3×3 è lo stesso dello strato con filtri 7×7 definito nella prima architettura. La differenza sostanziale però, è che tre strati permettono di includere altrettante funzioni di attivazione non lineari, anziché solo una.

L'architettura disegnata è composta da un totale di 15 strati, di cui 10 che coinvolgono parametri, ed è visibile in dettaglio in tabella 6.4. L'idea è che l'utilizzo continuativo di piccoli filtri fin dal primo strato aiuti a cogliere dettagli più fini, conducendo quindi a una ricostruzione più accurata delle *features*, sia di basso livello che di alto livello.

6.2.3 Architettura *Inception*

Le ultime architetture testate prevedono l'implementazione del modulo *Inception* (4.4), il quale permette di eseguire simultaneamente operazioni di convoluzione con filtri diversi sullo stesso volume di input, concatenando in output le mappe di attivazione ottenute. In questo caso si è deciso di imitare alla lettera l'architettura *GoogLeNet* di Szegedy *et al.* (2015), la quale prevede un primo *convolutional layer* con filtri di dimensione 7×7 seguito da una cascata di 9 moduli *inception*, più uno strato *fully connected* a terminare l'architettura. La rete è stata stimata sia nella sua versione originale che nella versione *Batch-Normalized*.

Rete	Accuratezza test set	Accuratezza sul dataset NPDI
VGG	97.3%	81.4%
Inception	96.7%	79.9%
Inception + BN	98.1%	83.0%
Inception + BN (*ensemble*)	98.3%	83.7%

Tabella 6.5: Percentuale di corretta classificazione delle reti sui due diversi test set.

La tabella 6.5 mostra i risultati ottenuti dalle ultime 3 architetture proposte, allenate sulle prime 100 000 (+ data augmentation) immagini del dataset, e testate sia sulle rimanenti 30 000 che sull'intero dataset NPDI. Il quarto modello è un *ensemble* di tre reti stimate a partire da una diversa inizializzazione dei parametri, con lo scopo di raggiungere minimi locali diversi sulla superficie di perdita (la classificazione finale è ottenuta tramite voto di maggioranza).

L'enorme discrepanza fra i risultati ottenuti nei diversi test set ha spinto a trovare una soluzione in grado di garantire una migliore generalizzazione. Inizialmente si è pensato di aiutare la rete nell'estrazione delle *features* attraverso le costruzione di sottoclassi (sempre tramite *Web Crawling*): nonostante si tratti di un problema di classificazione binaria la classe da riconoscere è infatti molto ampia e generale, dal momento che include una varietà molto estesa di situazioni. Questa idea è stata però rigettata in favore di una soluzione equivalente, ma molto più veloce ed elegante: allenare la rete su un generico dataset di dimensioni molto più grandi, e soprattutto composto da molte più classi, per poi continuare il training sul dataset in esame, ma solo degli ultimi strati *fully-connected*, congelando i parametri dei *convolutional layers* appresi in precedenza. L'idea è che allenando la rete su un dateset molto complesso si possa ottenere un'elevata capacità di estrazione delle *features*, capacità che appartiene ai *convolutional layers*, e che quindi può essere riutilizzata in altri dataset continuando la fase di training solo degli strati *fully connected*, i quali hanno il compito di mettere insieme le informazioni estratte ed effettuare la classificazione finale. In particolare, per la prima fase di training della rete è stato utilizzato il dataset ILSVRC1000 (Deng *et al.*, 2009), il quale comprende circa 500 000 immagini divise in 1000 diverse categorie; la scelta dell'architettura è ricaduta invece su quella che ha fornito i risultati migliori, ovvero la rete *Inception* con *Batch-Normalization*. L'allenamento della rete è stato fermato alla ventesima epoca, ed è stato continuato, previo congelamento dei parametri relativi ai *convolutional layers*, sul proprio dataset. I risultati ottenuti sono riportati in tabella 6.6: è stato registrato un sensibile guadagno in termini di generalizzazione, con un'errore di classificazione sul test set inferiore al punto percentuale e un accuratezza finale sul dataset NPDI molto soddisfacente. É bene infatti sottolineare che quest'ultimo valore è in realtà una sottostima dovuta alla presenza dei falsi positivi; inoltre, per poter essere utilizzate come test set, le immagini del dataset NPDI sono state portate ad avere un *aspect-ratio* (1:1) molto diverso da quello originale dei video dai quali sono state estratte (4:3), e pertanto presentano una notevole compressione orizzontale.

Rete	Accuratezza test set	Accuratezza sul dataset NPDI
Inception + BN + *pre-training*	99.1%	91.0%

Tabella 6.6: Performance del modello finale pre-allenato sul dataset ILSVRC1000.

Capitolo 7

Aspetti Computazionali

7.1 Configurazione software

La scelta del *framework* da utilizzare è ricaduta su *Mxnet* (Chen *et al.* (2015)) in quanto è l'unico a garantire pieno supporto al linguaggio R, ossia il linguaggio che inizialmente si era deciso di usare. Successivamente però si è rivelato necessario interfacciarsi al *framework* tramite un linguaggio di programmazione più efficiente, nello specifico *python*, in quanto la cattiva gestione della memoria di R limitava pesantemente la possibilità di costruire architetture più profonde.
Mxnet offre anche la possibilità di trasferire il carico computazionale sulla GPU, ma affinché questo sia possibile è necessario installare degli strumenti che permettano di interfacciarsi con la scheda grafica e di parallelizzare le operazioni dettate dalle istruzioni in linguaggio nativo del *framework*, ovvero c++. Questi strumenti sono il CUDA toolkit e il cuDNN (cuda Deep Neural Network library), i quali sono entrambi proprietari di Nvidia. Questo significa che GPU di altre marche, come ad esempio la concorrente AMD, non possono essere configurate per i *framework* di *deep learning*.
Le versioni di CUDA e cuDNN utilizzate in questa tesi sono rispettivamente la 8.0 e la 5.1. In realtà l'installazione del cuDNN non è indispensabile ai fini della corretta esecuzione dei comandi, ma promette un incremento in termini di velocità del 40% circa per operazioni di convoluzione e pooling, e pertanto è conveniente utilizzarlo se si deve lavorare con le *Convolutional Neural Networks*.
Per quanto riguarda la fase di preprocessing tutte le operazioni si sono svolte su ambiente *python* tramite l'utilizzo delle librerie PIL, scikit-image, OpenCV e OpenCV2.

7.2 Calcolo parallelo e sfruttamento della GPU

Nonostante siano state introdotte intorno agli anni '80 le reti neurali hanno cominciato ad emergere solamente dopo la metà degli anni duemila, quando l'incremento delle risorse computazionali ha permesso di fronteggiare il più grosso handicap di questi modelli, ovvero il grande carico computazionale. In particolare, oltre all'au-

mento generale delle risorse è stato anche il modo in cui questo è avvenuto che ha permesso alle reti neurali di emergere. Fino agli duemila infatti, la via seguita per incrementare le capacità di calcolo era quella di aumentare la frequenza di operazioni al secondo; quando però è stato raggiunto il limite fisico la scelta obbligata è stata quella di aumentare il numero di unità di calcolo (i *cores*), piuttosto che la frequenza, eseguendo quindi più operazioni in parallelo. Questa metodologia di calcolo si sposa benissimo con l'architettura delle reti neurali, dove il calcolo dei nodi appartenenti allo stesso strato può avvenire in maniera completamente indipendente. *Mxnet*, il *framework* utilizzato, prevede inoltre un altro livello di parallelizzazione, ovvero lungo le singole osservazioni che compongono il *batch*: sia lo stage *feed-forward* che quello di *back-propagation* vengono effettuati su unità di calcolo differenti per ogni osservazione, e il gradiente globale per l'aggiornamento dei pesi viene calcolato come media dei singoli gradienti (3.9). La motivazione per la quale le reti neurali vengono spesso allenate sulla GPU è che queste architetture permettono di sfruttare al meglio il calcolo parallelo: l'architettura tipica dei processori grafici è composta infatti da un numero elevato di unità di calcolo, mentre le normali CPU sono generalmente dual-core o quad-core. Inoltre, mentre quest'ultime sono pensate per operazioni di calcolo generali, le GPU si adattano meglio a svolgere operazioni vettoriali e matriciali, ossia le stesse che avvengono durante la fase *feed-forward* delle reti neurali, e soprattutto durante le operazioni di convoluzione delle *Convolutional Neural Networks.*
In tabella 7.1 si possono notare le differenze prestazionali fra le diverse architetture utilizzate, durante una fase di training della rete *Inception-BN* sull'intero dataset (130 000 immagini).

Processore	Unità di calcolo	Frequenza	Immagini/sec	Tempo/epoca
CPU intel i5-4210u	4	2.7 Ghz	0.6	62h
GPU Nvidia 840M	384	1.0 Ghz	7.1	5h14m
GPU Nvidia GTX980	2048	1.2 Ghz	43.7	51m

Tabella 7.1: Prestazioni di diverse architetture CPU e GPU sulla rete *Inception-BN*

7.3 *Iterator* e determinazione del *batch size*

L'unico svantaggio di eseguire il lavoro computazionale sulla GPU piuttosto che sulla CPU è relativo alla memoria RAM. Mentre la memoria della CPU è espandibile fino a 32 o 64 GigaByte la memoria delle GPU è fissa e generalmente poca. In questa tesi si è utilizzata inizialmente una GPU Nvidia 840M, la quale mette a disposizione 2 GigaByte di RAM, mentre nelle fasi successive si è usata una GTX980 che offre invece 4 GigaByte. Chiaramente entrambe queste quantità non sono sufficienti ad immagazzinare tutte le 130 000 immagini che compongono il dataset, ma se si utilizza un ottimizzatore *Stochastic Gradient Descent* questo problema può essere facilmente aggirato caricando in memoria solamente un *batch* di immagini per volta,

ovvero quelle necessarie in quella particolare fase di stima. A livello implementativo questo è possibile attraverso la definizione di un iteratore, il quale opera andando a leggere sul disco fisso il *batch* di immagini richiesto dalla rete e caricandolo in memoria RAM dopo averlo decompresso nelle matrici di numeri grezze, dal momento che le immagini sono salvate in JPEG. Affinché l'iteratore riesca a tenere il passo delle richieste che giungono dalla rete è necessario compattare le immagini un unico grande file, in modo che venga sfruttata la velocità di lettura sequenziale del disco fisso, il quale non ha le stesse prestazioni della memoria volatile.

Strato	**Volume di input**	**Tot. neuroni**	**Numero di parametri**
Immagine	$3 \times 224 \times 224$	150 528	0
Conv_3_64	$224 \times 224 \times 64$	3 211 264	$(3 \times 3 \times 3) \times 64 = 1\,728$
Conv_3_64	$224 \times 224 \times 64$	3 211 264	$(3 \times 3 \times 64) \times 64 = 36\,864$
Pool_2	$112 \times 112 \times 64$	802 816	0
Conv_3_128	$112 \times 112 \times 128$	1 605 632	$(3 \times 3 \times 64) \times 128 = 73\,728$
Conv_3_128	$112 \times 112 \times 128$	1 605 632	$(3 \times 3 \times 128) \times 128 = 147\,456$
Conv_3_256	$56 \times 56 \times 256$	802 816	$(3 \times 3 \times 128) \times 256 = 294\,912$
Conv_3_256	$56 \times 56 \times 256$	802 816	$(3 \times 3 \times 256) \times 256 = 589\,824$
Conv_3_256	$56 \times 56 \times 256$	802 816	$(3 \times 3 \times 256) \times 256 = 589\,824$
Pool_2	$28 \times 28 \times 256$	200 704	0
Conv_3_512	$28 \times 28 \times 512$	401 408	$(3 \times 3 \times 256) \times 512 = 1\,179\,648$
Conv_3_512	$28 \times 28 \times 512$	401 408	$(3 \times 3 \times 512) \times 512 = 2\,359\,296$
Conv_3_512	$28 \times 28 \times 512$	401 408	$(3 \times 3 \times 512) \times 512 = 2\,359\,296$
Pool_2	$14 \times 14 \times 512$	100 352	0
Conv_3_512	$14 \times 14 \times 512$	100 352	$(3 \times 3 \times 512) \times 512 = 2\,359\,296$
Conv_3_512	$14 \times 14 \times 512$	100 352	$(3 \times 3 \times 512) \times 512 = 2\,359\,296$
Conv_3_512	$14 \times 14 \times 512$	100 352	$(3 \times 3 \times 512) \times 512 = 2\,359\,296$
Pool_2	$7 \times 7 \times 512$	25 088	0
FC	$1 \times 1 \times 2048$	2 048	$7 \times 7 \times 512 \times 2048 = 51\,380\,224$
FC	$1 \times 1 \times 2048$	2 048	$2048 \times 2048 = 4\,194\,304$
Totale		14 831 104	70 284 992

Tabella 7.2: Calcolo del numero di parametri e di neuroni necessario per ogni strato della rete. I due numeri dopo l'underscore indicano rispettivamente estensione spaziale e numero dei filtri.

La limitata disponibilità di RAM si riflette anche sulla numerosità massima che possiamo scegliere per il *batch*, la quale non è dettata solamente dal peso della singola immagine: bisogna tenere conto infatti anche dello spazio necessario ad immagazzinare neuroni e parametri della rete. A titolo esemplificativo vengono riportati in tabella 7.2 i calcoli per la determinazione di quest'ultimi, nel caso della seconda architettura utilizzata (6.2.2). In totale la rete richiede un costo fisso di oltre 70 milioni di parametri, che allocati in celle di memoria della dimensione di 4 bytes ognuna si traducono in 275 MegaBytes; Dal momento che la rete viene parallelizzata sulle singole osservazioni ogni immagine richiede invece l'utilizzo di quasi 15 milioni

di neuroni, solo per la fase *feed-forward*, più altrettanti valori necessari ad immagazzinare i gradienti durante la fase di *back-propagation*: tradotto in bytes, ogni immagine comporta il consumo di circa 120 Mega. Dal totale di memoria RAM disponibile bisogna quindi sottrarre i 275 MegaBytes riservati ai parametri, 200/300 MegaBytes riservati all'interfaccia grafica del PC, e i rimanenti possono essere utilizzati per la determinazione della numerosità massima del *batch*, dividendoli per il consumo comportato dalla singola immagine. Nel caso dell'architettura in esame, una GPU con 4 GigaBytes come quella utilizzata in questa tesi consente quindi di utilizzare fino a 27 immagini per volta prima di effettuare l'aggiornamento dei pesi.

Capitolo 8

Conclusioni

La sfida all'origine di questa tesi è stata quella di costruire un modello in grado di riconoscere in maniera completamente autonoma la presenza di nudità nelle immagini. Nonostante questo si traduca in un problema di classificazione binaria la complessità di tale compito è più grande di quello che si possa pensare: la classe da riconoscere comprende infatti una varietà infinita di possibili situazioni che si possono presentare, e di conseguenza risulta difficile da generalizzare. Questo, unito alla scarsa qualità dei dataset disponibili online, ha portato al primo problema da affrontare, ovvero la costruzione di un dataset che fosse sufficientemente rappresentativo. A tal scopo ci si è serviti della pratica del Web Crawling, la quale viene spesso affiancata alle attività di data mining proprio perchè permette di raccogliere grosse quantità di dati in breve tempo.
La grande mole di immagini da analizzare si è riflessa nel secondo grande ostacolo, ovvero rendere il problema sostenibile dal punto di vista computazionale. Qui si è arrestata la corsa della maggior parte dei tradizionali metodi statistici, mentre le *Convolutional Neural Networks* hanno saputo offrire maggiore flessibilità grazie alla possibilità si essere stimate in maniera ricorsiva e alla loro propensione al calcolo parallelo.
Aldilà degli aspetti computazionali, i metodi statistici utilizzati (*gradient boosting*) hanno mostrato i loro limiti anche sotto il profilo teorico, non mostrandosi in grado di cogliere le *features* discriminanti: la loro invarianza rispetto a traslazione risulta infatti un grande handicap nel campo del riconoscimento di immagini, e pertanto per incrementare le loro performance è necessario ricorrere all'estrazione manuale delle *features* tramite operazioni di *pre-processing*.
Le proprietà di interazione sparsa, invarianza rispetto a traslazione e condivisione dei parametri su cui poggiano le *Convolutional Neural Networks* fanno di loro un potente strumento di classificazione di immagini, campo in cui rappresentano al giorno d'oggi lo stato dell'arte. Il grande svantaggio risiede però nell'accentuata propensione al sovradattamento: il loro potenziale viene espresso solamente se alimentate da una sufficiente quantità di dati, che spesso non è disponibile o risulta difficile da collezionare per l'utente privato. In ogni caso, i risultati ottenuti in questa tesi con l'utilizzo di un dataset relativamente piccolo sono rilevanti e soprattutto promet-

tenti per quei soggetti come facebook, google, instagram, o in generale piattaforme di condivisione di immagini, che dispongono di una fonte quasi illimitata di osservazioni su cui possono facilmente costruire, a partire da una *Convolutional Neural Network*, un classificatore che non solo rappresenta lo stato dell'arte, ma che può addirittura superare le capacità dell'uomo.
Possibili sviluppi futuri in grado di incrementare ulteriormente le performance del classificatore potrebbero prevedere l'utilizzo di eventuali informazioni aggiuntive disponibili, quali il testo se si tratta di immagini caricate sui social network, oppure la traccia audio nel caso di immagini provenienti da filmati.
Un'altra possibile strada potrebbe essere invece lo sviluppo di una rete in grado di apprendere in maniera autonoma dai propri errori, tramite la combinazione di metodi supervisionati e non.

Bibliografia

Arora S. *et al.* (2014). Provable bounds for learning some deep representations.

Bing Xu, Naiyan Wang T. C. M. L. *et al.* (2015). Empirical evaluation of rectified activations in convolution network.

Chen T.; Guestrin C. (2016). Xgboost: A scalable tree boosting system. *arXiv preprint arXiv:1603.02754*.

Chen T.; Li M.; Li Y.; Lin M.; Wang N.; Wang M.; Xiao T.; Xu B.; Zhang C.; Zhang Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Deng J.; Dong W.; Socher R.; Li L.-J.; Li K.; Fei-Fei L. (2009). Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255. IEEE.

Duchi J.; Hazan E.; Singer Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, **12**(Jul), 2121–2159.

Efron B.; Hastie T. (2016). *Computer Age Statistical Inference: Algorithms, Evidence, and Data Science*.

Freund Y.; Iyer R.; Schapire R. E.; Singer Y. (1997). An efficient boosting algorithm for combining preferences. *Journal of machine learning research*, **4**(Nov), 933–969.

Friedman J.; Hall P. (2000). On bagging and nonlinear estimation.

Hastie T.; Tibshirani R.; Friedman J. (2009). *The elements of statistical learning*. Springer New York.

Ian Goodfellow, Yoshua Bengio A. C. (2016). *Deep Learning*. MIT Press, draft of august 10, 2016 edizione.

Karpathy A. (2015). Stanford university cs231n: Convolutional neural networks for visual recognition.

Kingma D. P.; Ba J. (2014). Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*.

Krizhevsky A.; Sutskever I.; Hinton G. E. (2012). Imagenet classification with deep convolutional neural networks In *Advances in Neural Information Processing Systems 25*. A cura di Pereira F., Burges C. J. C., Bottou L., Weinberger K. Q., pp. 1097–1105. Curran Associates, Inc.

Ridgeway G. *et al.* (2006). gbm: Generalized boosted regression models. *R package version*, **1**(3), 55.

Ruder S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, **1609.04747**.

Simonyan K.; Zisserman A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Srivastava N.; Hinton G. E.; Krizhevsky A.; Sutskever I.; Salakhutdinov R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, **15**(1), 1929–1958.

Szegedy C.; Liu W.; Jia Y.; Sermanet P.; Reed S.; Anguelov D.; Erhan D.; Vanhoucke V.; Rabinovich A. (2015). Going deeper with convolutions. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Taigman Y.; Yang M.; Ranzato M.; Wolf L. (2014). Deepface: Closing the gap to human-level performance in face verification. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, p. 8. IEEE.

Valiant L. G. (1984). A theory of the learnable. *Commun. ACM*, **27**(11), 1134–1142.

Xavier (2010). Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pp. 249–256.