UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Classificazione neurale su microcontrollori di suoni ambientali eterogenei

Tiny Neural Classification on microcontroller of heterogeneus environmental sounds

Candidato:
Davide Pio Ciavarella

Relatore:
Prof. Claudio Turchetti

Anno Accademico 2020-2021

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Classificazione neurale su microcontrollori di suoni ambientali eterogenei

Tiny Neural Classification on microcontroller of heterogeneus environmental sounds

Candidato:
Davide Pio Ciavarella

Relatore:
Prof. Claudio Turchetti

Anno Accademico 2020-2021

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica
Via Brecce Bianche – 60131 Ancona (AN), Italy

Indice

1 Introduzione **1**

1.1 Organizzazione della tesi 1

2 Neural Network **3**

2.1 Motivazione biologica 3

2.2 Architettura di una rete neurale 4

2.3 Stima di una rete neurale 6

2.3.1 Calcolo del gradiente : l'algoritmo di *back propagation* 7

2.3.2 Stochastic Gradient Descent (SGD) 8

2.3.3 Algoritmi di ottimizzazione della discesa del gradiente 9

2.3.4 Forme di regolarizzazione 14

2.4 Funzioni di non linearità 17

2.4.1 Sigmoide 17

2.4.2 Tangente iperbolica 19

2.4.3 ReLU 20

2.4.4 Leaky ReLU 21

2.5 Inizializzazione dei pesi 21

2.5.1 Batch Normalization 23

3 Convolution Neural Network **27**

3.1 L'operazione di convoluzione 28

3.2 Strati di una CNN 31

3.2.1 Convolution Layer 31

3.2.2 Pooling Layer 33

3.2.3 Fully Connected Layers 34

3.3 Architettura generale della rete 34

3.4 Data augumentation 36

4 Dataset **37**

4.1 ESC 37

4.2 Pre-processing 38

4.2.1 Spettrogrammi 39

4.2.2 Mel-spectrogram 42

4.2.3 Divisione spettrogrammi 43

4.3 Data augumentation 46

5 Analisi della rete neurale proposta **49**
5.1 Convolution Neural Network 49
5.2 Iperparametri 51
5.3 K-fold validation 52
5.4 Risultati training 52
5.5 Post-training quantization 54
5.6 X-Cube AI 56

6 Dimostratore **61**

7 Conclusioni e sviluppi futuri **65**

Elenco delle figure

2.1 Struttura di un neurone 3
2.2 Struttura matematica di un neurone 4
2.3 Architettura di una rete neurale con un solo strato nascosto 5
2.4 Architettura di una rete neurale con più strati nascosti 5
2.5 Fluttuazione SGD con dimensione batch pari ad uno 9
2.6 Discesa del gradiente senza (a) e con (b) ottimizzazione "momentum" 10
2.7 Mentre il *momentum* calcola prima il gradiente (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). 11
2.8 Performance degli algoritmi di ottimizzazione SGD sui livelli di una superficie di perdita. Fonte: Ruder (2016). 14
2.9 Performance degli algoritmi di ottimizzazione SGD sui livelli di una superficie di perdita. Fonte: Ruder (2016). 14
2.10 Contorni della regione imposta dalla (2.26), per differenti valori di q 15
2.11 Contorni della regione imposta dalla (2.27) con q=2 (sinistra), elastic-net con $\alpha = 0.2$(destra). Nonostante i due contorni appaiono simili, solo l'elastic-net risulta non differenziabile negli angoli. 16
2.12 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. 16
2.13 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. 17
2.14 Funzione sigmoide. 18
2.15 Esempio di dinamica a zig-zag nell'aggiornamento di due pesi W1 e W2. 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. 19
2.16 Andamento tangente iperbolica. 19
2.17 Andamento ReLU. 20
2.18 Andamento Leaky ReLU. 21

3.1 Operazione di convoluzione nel caso bidimensionale: un filtro di dimensione $2x2$ viene moltiplicato elemento per elemento per una porzione dell'input di uguali dimensioni.n 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. 28
3.2 Operazione di convoluzione di un filtro di dimensioni $5x5$ e relativa mappa di attivazioni prodotta. 29
3.3 Operazione di convoluzione di due differenti filtri ($W0$ e $W1$) di dimensione $3x3x3$ su un volume di input $7x7x3$. 30
3.4 Set di 96 filtri 11 x 11 x 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. 33
3.5 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). 34
3.6 Esempio di apprendimento delle features su un dataset di oggetti misti. 35

4.1 Suoni nel tempo 38
4.2 Trasformata di Fourier 39
4.3 Suoni nel dominio della frequenza 40
4.4 Calcolo delle FFT per spettrogramma 41
4.5 Finestra di Hamming 42
4.6 Grafico funzione Mel 43
4.7 Spettrogrammi degli audio di Figura 4.1 44
4.8 Finestratura spettrogramma 46

5.1 Convolution Neural Network proposta. 50
5.2 Learning rate variabile. 51
5.3 Divisione del dataset per K-fold validation 52
5.4 Loss ed accuracy del training 53
5.5 Confusion matrix 54
5.6 Processo post-training quantization 55
5.7 Setting post-training quantization 55
5.8 Integer quantization (int8/asymmetric) 56
5.9 X-Cube AI overwiev 57
5.10 Report X-Cube AI 58

6.1 Tempo per classificazione di un audio 63

Elenco delle tabelle

4.1 Divisione ESC-50. 37
4.2 Dettagli ESC-10 47

5.1 Architettura rete neurale 49
5.2 Confronto risultati 53
5.3 Parametri delle CNN' 57
5.4 Accuracy e inference time della CNN 58

Capitolo 1

Introduzione

Il riconoscimento del suono è un argomento centrale nelle teorie odierne sul riconoscimento dei modelli, che copre una ricca varietà di campi. Alcuni degli argomenti relativi al riconoscimento del suono hanno fatto notevoli progressi nella ricerca, come il riconoscimento automatico del parlato (ASR) [1, 2] e il recupero delle informazioni musicali (MIR) [3, 4]. La classificazione del suono ambientale (ESC) è un altro importante ramo del riconoscimento del suono ed è ampiamente applicata nella sorveglianza [5], nell'automazione domestica [6], nell'analisi delle scene [7] e nell'udito delle macchine [8].
Tuttavia, a differenza del parlato e della musica, gli eventi sonori sono più diversi con un'ampia gamma di frequenze e spesso meno ben definiti, il che rende le attività ESC più difficili di ASR e MIR. Pertanto, ESC deve ancora affrontare problemi di progettazione critici in termini di prestazioni e miglioramento della precisione.
In questo lavoro di tesi andremo a vedere un modello di rete neurale convolutiva per l'estrazione di features da spettrogrammi; e infine andremo a creare un dimostratore di riconoscimento di suoni ambientali eterogenei che opera su un micro controllore.

1.1 Organizzazione della tesi

Per rendere la tesi leggibile, è stata divisa in capitoli presentati in questo modo :

- **Capitolo 2**. Andremo a fare una descrizione di quella che è la teoria delle reti neurali.
- **Capitolo 3**. In questo capitolo si va a descrivere cosa è una *Convolution Neural Network* e perchè sono così fondamentali per le reti neurali odierne.
- **Capitolo 4**. In questa sezione si parlerà del dataset usato e di come sia stato fatto il pre-processing, oltre al data augumentation.
- **Capitolo 5**. Approfondiremo dei concetti un pò più pratici, inerenti alla rete scelta, al relativo training, e di come la rete addestrata venga modificata per lavorare sui micro controllori.
- **Capitolo 6**. Discuteremo del dimostratore costruito, come lavora e il tempo che impiega per effettuare la classificazione di un suono ambientale.

- **Capitolo 7**. L'ultimo capitolo riguarda le conclusioni e le considerazioni su un possibile sviluppo futuro.

Capitolo 2

Neural Network

Prima di iniziare ad entrare nello studio delle Convolution Neural Network è bene tornare indietro e conoscere la storia delle reti neurali. Le reti neurali, nate guardando l'architettura del cervello umano, sono dei modelli ad elevato numero di parametri che se alimentati con un gran numero di dati, sono in grado di apprendere delle relazioni predittive. Inventate intorno agli anni '80, furono abbandonate per diversi decenni a causa della scarsa teoria e della risorsa tecnologia scarseggiante. Nei primi anni 2000 sono tornate di auge grazie al notevole miglioramento sia della parte teorica, ma anche del netto miglioramento delle risorse computazionali. Questo ha decretato l'inizio del deep learning, con la prospettiva di risolvere dei compiti molto difficili come la classificazione di immagini e video, ma anche al riconoscimento vocale o testuale.

2.1 Motivazione biologica

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 da circa 86 miliardi, connessi fra di loro attraverso un numero di sinapsi dell'ordine di 10^{15}.

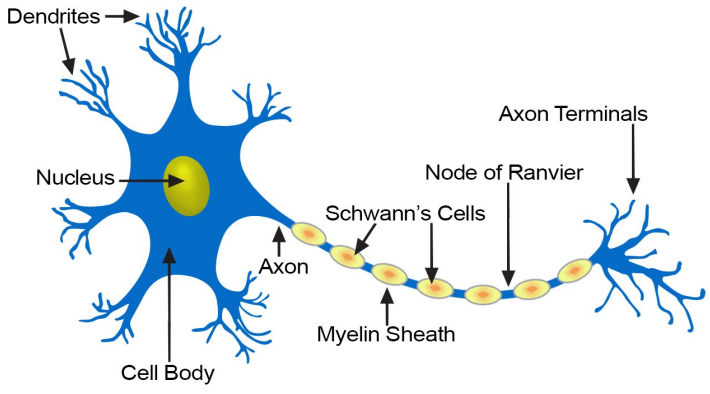


Figura 2.1: Struttura di un neurone

La Figura 2.1 mostra la rappresentazione biologica di un neurone; ogni neurone riceve in input il segnale dai dendriti e, una volta che il nucleo lo ha elaborato, produce un segnale in output lungo il suo unico assone, che lo collega ai neuroni successivi attraverso le sinapsi.
Matematicamente, il neurone può essere visto come in Figura 2.2. I segnali viaggiano attraverso gli assoni x_0 ed attraverso un prodotto w_0x_0 interagiscono con i dendriti con i quali sono collegati attraverso le sinapsi w_0.

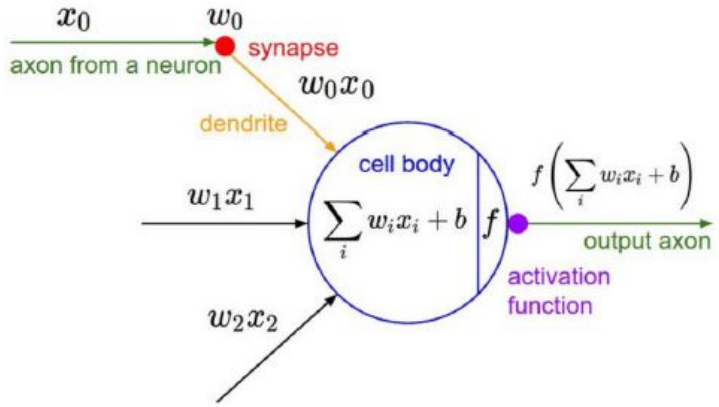


Figura 2.2: Struttura matematica di un neurone

L'idea è che l'insieme delle sinapsi *(i pesi w)* possa in qualche modo essere 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 supera una certa soglia, il neurone invierà un impulso attraverso il suo assone. Tale somma viene modellata da una funzione di attivazione, che tipicamente coincide con la funzione sigmoide

$$\sigma = \frac{1}{1 + e^{-x}} \tag{2.1}$$

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).

2.2 Architettura di una rete neurale

In Figura 2.3 è rappresentato il diagramma di una semplice rete neurale con quattro ingressi x_j, uno strato nascosto composto da cinque neuroni $a_l = g(w_{l_0}^{(1)} + \sum_{j=1}^{4} w_{lj}^{(1)} x_j)$ e una singola 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 dei pesi $[w_{lj}^{(1)}]$, dove l'(1) si riferisce al primo strato, j al j'esimo ingresso e l all'l-esimo neurone. g è

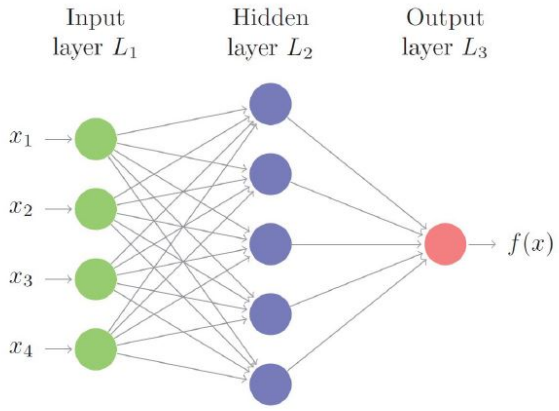


Figura 2.3: Architettura di una rete neurale con un solo strato nascosto

la funzione di attivazione descritta in precedenza. L'idea di base è quella che ogni neurone apprenda una semplice funzione binaria on/off, 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.
In Figura 2.4 vediamo rappresentata una rete neurale con più strati nascosti (in questo caso tre), e con dieci nodi di output (uno per classe). Queste reti costituiscono il *deep learning* proprio per la profondità in termine di numero di strati nascosti che le compongono.

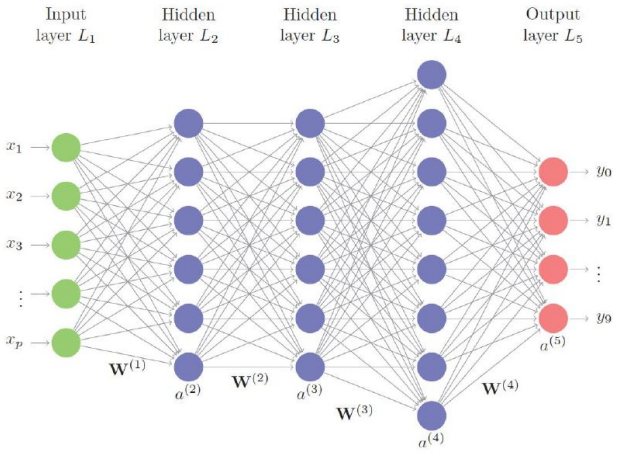


Figura 2.4: Architettura di una rete neurale con più strati nascosti

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

$$z_l^{(2)} = w_{l_0}^{(1)} + \sum_{j=1}^{p} w_{lj}^{(1)} x_j \tag{2.2}$$
$$a_l^{(2)} = g^{(2)}(z_l^2)$$

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 :

$$z_l^{(k)} = w_{l_0}^{(k-1)} + \sum_{j=1}^{p} w_{lj}^{(k-1)} a_j^{(k-1)} \tag{2.3}$$
$$a_l^{(k)} = g^{(k)}(z_l^k)$$

o equivalentemente in termini matriciali :

$$z^{(k)} = \mathbf{W}^{(k-1)} a^{(k-1)} \tag{2.4}$$
$$a^{(k)} = g^{(k)} z^{(k)}$$

dove $\boldsymbol{W}^{(k-1)}$ rappresenta la matrice dei pesi che vanno dallo strato L_{k-1} allo strato L_k, mentre $a^{(k)}$ è il vettore delle attivazione dello strato L_k. Per problemi di classificazione in M classi viene generalmente preferita come funzione di attivazione per l'ultimo strato la funzione *softmax*

$$g^{(K)} = \frac{e^{z_m^{(K)}}}{\sum_{l=1}^{M} e^{z_l^{(K)}}} \tag{2.5}$$

dove M è il numero delle classi.
La funzione *softmax* restituisce una probabilità per ogni classe, e la somma totale sarà sempre pari a 1.

2.3 Stima di una rete neurale

Stimare una rete neurale non risulta essere un compito semplice, trattandosi di una complessa funzione gerarchica *f(x;W)* del vettore di input x e della collezione di pesi *W*. Inanzitutto bisogna scegliere opportunamente le funzioni di attivazioni $g^{(k)}$ in modo che tale funzioni risulti differenziabile. Si tratta quindi di risolvere, dato un

set di osservazioni $[x_i, y_i]$ e una funzione di perdita L[*y,f(x)*], un problema di minimizzazione :

$$\min_W \left[\frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x_i; W)) + \lambda J(W) \right] \tag{2.6}$$

dove J(W) è un termine non negativo di regolarizzazione sugli elementi di W, con $\lambda relativoparametrodiregolarizzazione$.
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 richiedere molto tempo, che spesso non è disponibile, pertanto generalmente ci si tende ad accontentare di buoni minimi locali.

2.3.1 Calcolo del gradiente : l'algoritmo di *back propagation*

I principali metodi utilizzati per la risoluzione della (2.6) 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 (-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 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 :

$$\Delta W^{(k)} = \frac{1}{n} \sum_{i=1}^{n} = \frac{\partial L\left[y, f(x, W)\right]}{\partial W^{(k)}} \tag{2.7}$$

Algorithm 1 Back propagation

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 memoria le quantità calcolate).

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

$$\delta_l^{(K)} = \frac{\partial L[y, f(x,W)]}{\partial z_l^{(K)}} = \frac{\partial L[y, f(x,W)]}{\partial a_l^{(K)}} \dot{g}^{(K)}(z_l^{(K)}) \tag{2.9}$$

dove $\dot{g}$ denota la derivata prima di g 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 :

$$\delta_l^{(k)} = \left(\sum_{j=1}^{p_{k+1}} w_{jl}^{(k)} \delta_j^{(k+1)} \right) \dot{g}\left(z_l^{(k)}\right) \tag{2.10}$$

4: Le derivate parziali sono date da :

$$\frac{\partial L\left[y, f(x,W)\right]}{\partial w_{lj}^{(k)}} = a_j^{(k)} \delta_l^{(k+1)} \tag{2.11}$$

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

$$W^{(k)} \leftarrow W^{(k)} - \alpha(\Delta W^{(k)} + \lambda W^{(k)}), k = 1, ..., K-1 \tag{2.8}$$

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 (2.8) 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 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.

2.3.2 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 (2.8) 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 2.5). Di conseguenza si tendono a scegliere dei valori intermedi, tipicamente nel range [50,256].

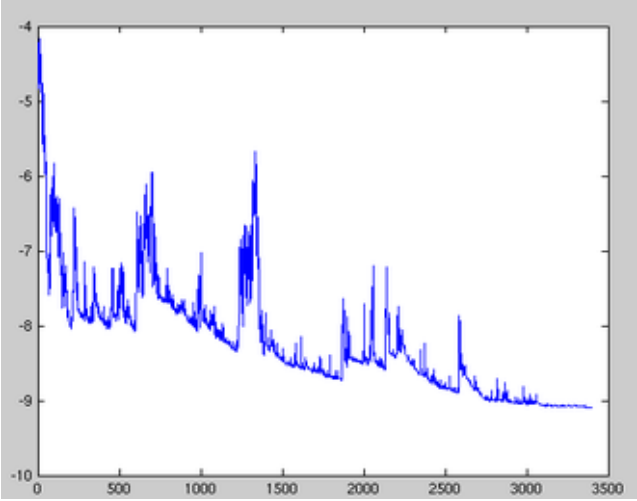


Figura 2.5: Fluttuazione SGD con dimensione batch pari ad uno

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).

2.3.3 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*.
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 2.6).

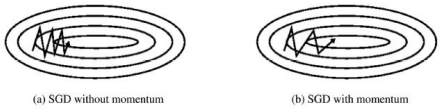


Figura 2.6: Discesa del gradiente senza (a) e con (b) ottimizzazione "momentum"

Il *momentum* è un metodo che aiuta ad accellerare la discesa del gradiente verso la direzione corretta, smorzando l'effetto indotto dalle oscillazioni. Tale risultato si ottiene aggiungendo al termine di aggiornamento corrente una funzione γ del vettore di aggiornamento precedente :

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_W \varphi(W^{(K)}) \\ W^{(k)} &\leftarrow W^{(k)} - v_t \end{aligned} \tag{2.12}$$

dove $\nabla_W \varphi(W^{(K)}$ è il gradiente rispetto a W della funzione da minimizzare (argomento della (2.6)). Tipicamente vengono utilizzati valori di γ nell'intorno di 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 rispetto alla previsione della posizione futura:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \alpha \nabla_W \varphi(W^{(K)} - \gamma v_{t-1}) \\ W^{(k)} &\leftarrow W^{(k)} - v_t \end{aligned} \tag{2.13}$$

In fig. Figura 2.7 vediamo la differenza tra *momentum e NAG*.

Adagrad

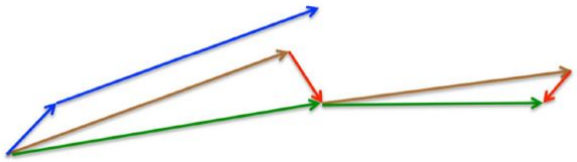


Figura 2.7: Mentre il *momentum* calcola prima il gradiente (freccia piccola blu) per poi saltare in direzione del gradiente accumulato (freccia grande blu), il NAG esegue prima un salto in direzione del gradiente accumulato in precedenza (freccia marrone), e successivamente calcola il gradiente corrente, applicando una correzione (freccia verde).

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 $w_{lj}^{(k)}$ ad ogni iterazione t, modificando quest'ultimo sulla base dei gradienti passati che sono stati calcolati per lo specifico parametro:

$$w_{t+1,lj}^{(k)} = w_{t,lj}^{(k)} - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} \nabla_w \varphi(w_{t,lj}^{(k)}) \tag{2.14}$$

o equivalentemente in termini matriciali :

$$W_{t+1}^{(k)} = W_t^{(k)} - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \nabla_W \varphi(W_t^{(k)}) \tag{2.15}$$

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 $w_{lj}^{(k)}$ 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. Infatti, 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 :

$$E\left[\nabla_W \varphi(W^{(k)})^2\right]_t = \gamma E\left[\nabla_W \varphi(W^{(k)})^2\right]_{t-1} + (1-\gamma)\nabla_W \varphi(W_t^{(k)})^2 \tag{2.16}$$

La (2.15) diventa quindi, in termini di aggiornamento :

$$\begin{aligned} \Delta W_t^{(k)} &= -\frac{\alpha}{\sqrt{E\left[\nabla_W \varphi(W^{(k)})^2\right]_t + \epsilon}} \nabla_W \varphi(W_t^{(k)}) \\ &= -\frac{\alpha}{RMS\left[\nabla_W \varphi(W^{(k)})\right]_t} \nabla_W \varphi(W_t^{(k)}) \end{aligned} \tag{2.17}$$

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:

$$E\left[\Delta W^2\right]_t = \gamma E\left[\Delta W^2\right]_{t-1} + (1-\gamma)\Delta W_t^2 \tag{2.18}$$

con errore quadratico medio :

$$RMS\left[\Delta W^2\right]_t = \sqrt{E\left[\Delta W^2\right]_t + \epsilon} \tag{2.19}$$

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

$$\begin{aligned} \Delta W_t^{(k)} &= -\frac{RMS\left[\Delta W^{(k)}\right]_{t-1}}{RMS\left[\nabla_W \varphi(W^{(k)})\right]_t} \nabla_W \varphi(W_t^{(k)}) \\ W_{t+1}^{(k)} &= W_t^{(k)} + \Delta W_t^{(k)} \end{aligned} \tag{2.20}$$

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 :

$$m_t = \beta_1 m - t - 1 + (1 - \beta_1)\nabla_W \varphi(W_t^{(k)}) \tag{2.21}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2 \left[\nabla_W \varphi(W_t^{(k)})\right]^2 \tag{2.22}$$

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 ($\beta_1 e \beta_2$ prossimi all'unità), di conseguenza, nella regola di aggiornamento viene utilizzata la loro stima corretta:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.23}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.24}$$

$$W_{t+1}^{(k)} = W_t^{(k)} - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2.25}$$

I valori di default per l'algoritmo Adam 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 Figura 2.8 gli algoritmi seguono tutti un percorso diverso l'uno dall'altro per raggiungere il minimo.
Adagrad e Adadelta 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 leggera modifica del numeratore.
In figura Figura 2.9 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

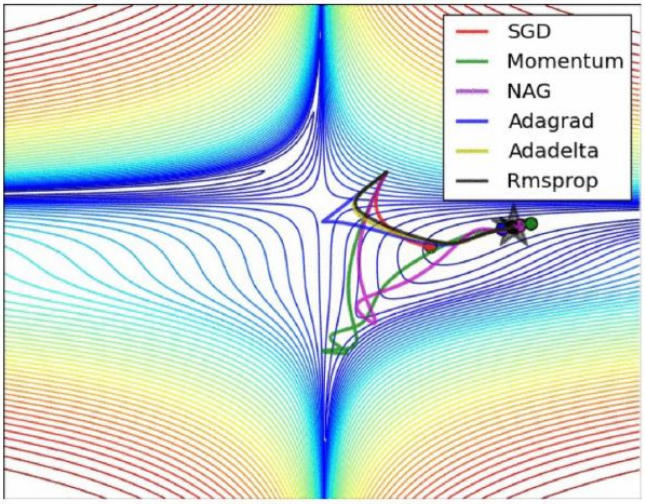


Figura 2.8: Performance degli algoritmi di ottimizzazione SGD sui livelli di una superficie di perdita. Fonte: Ruder (2016).

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.

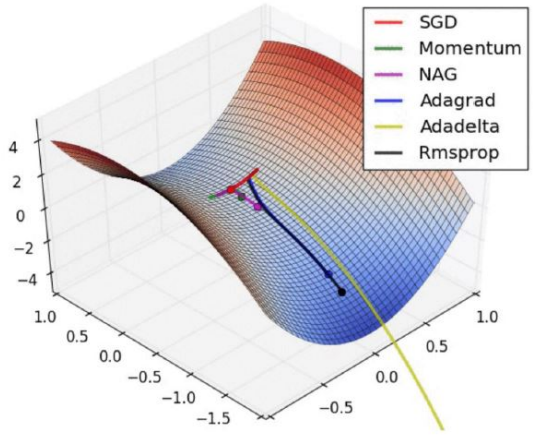


Figura 2.9: Performance degli algoritmi di ottimizzazione SGD sui livelli di una superficie di perdita. Fonte: Ruder (2016).

2.3.4 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 (2.6)) che vanno a regolare l'influenza dei vari parametri all'interno del modello.

Penalizzazioni

Generalmente, il termine di penalizzazione ha una forma del tipo :

$$J(W) = \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{l=1}^{p_k-1} \left[|w_{lj}^{(k)}| \right]^q \tag{2.26}$$

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).
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.

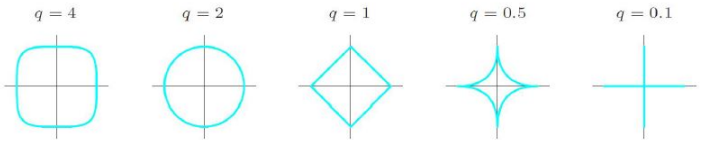


Figura 2.10: Contorni della regione imposta dalla (2.26), per differenti valori di q

Un valido compromesso fra le due forme di penalizzazione è rappresentato dall' *elasticnet* :

$$J(W) = \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{l=1}^{p_k-1} \left(\alpha (w_{lj}^{(k)})^q + (1-\alpha) |w_{lj}^{(k)}| \right) \tag{2.27}$$

la quale, presentando una regione con angoli non differenziabili, mantiene la caratteristica del lasso di settare i coefficienti esattamente a zero, ma in maniera molto più moderata, in relazione alla scelta di α.
Un esempio è visibile in fig. Figura 2.11.

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 Figura 2.12).

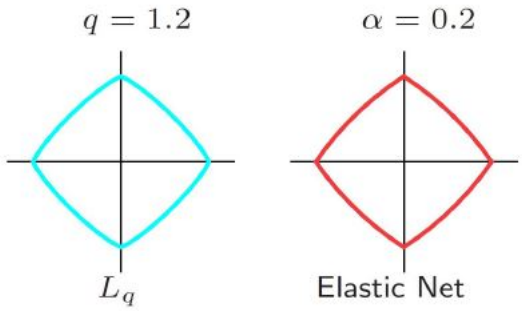


Figura 2.11: Contorni della regione imposta dalla (2.27) con q=2 (sinistra), elastic-net con $\alpha = 0.2$(destra). Nonostante i due contorni appaiono simili, solo l'elastic-net risulta non differenziabile negli angoli.

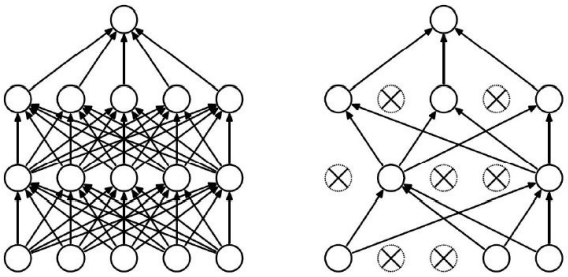


Figura 2.12: Diagramma di una rete neurale prima (sinistra) e dopo (destra) l'applicazione del dropout. I nodi segnati con una croce sono quelli che sono stati scelti casualmente per essere rimossi.

Formalmente, la transizione dallo strato *k-1* allo strato k (2.2) diventa :

$$\begin{aligned} r_j^{k-1} &\sim Bernoulli(1-\phi) \\ \widetilde{a}^{k-1} &= a^{k-1} \odot r_j^{k-1} \\ z_l^{(k)} &= w_{l_0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{lj}^{(k-1)} \widetilde{a}_j^{k-1} \\ a_l^{(k)} &= g^{(k)}\left(z_l^{(k)}\right) \end{aligned} \tag{2.28}$$

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).
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 2.13).

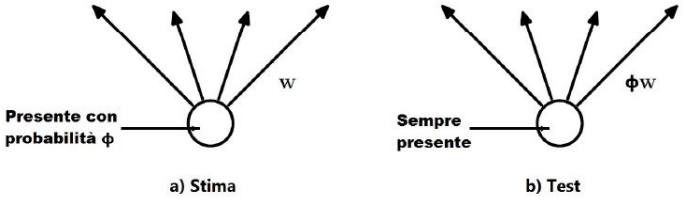


Figura 2.13: Sinistra : il neurone durante la fase di stima è presente con probabilità ϕ, ed è connesso con i neuroni dello strato successivo attraverso i pesi w. Destra: lo stesso neurone in fase di test è sempre presente nella rete e i suoi pesi sono moltiplicati per ϕ.

2.4 Funzioni di non linearità

2.4.1 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. Per capire il motivo di tutto ciò riportiamo innanzitutto l'esepressione matematica, e in Figura 2.14 la sua rappresentazione grafica.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.29}$$

La motivazione più importante che ha portato all suo accantonamento 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 :

$$\begin{aligned} \sigma(z)_{z\to\infty} &= 1 \\ \sigma(z)_{z\to-\infty} &= 0 \end{aligned} \tag{2.30}$$

Tale saturazione diventa problematica durante le fase di back propagation, in quanto il gradiente locale assume valori prossimi allo zero (2.31), che per la regola della

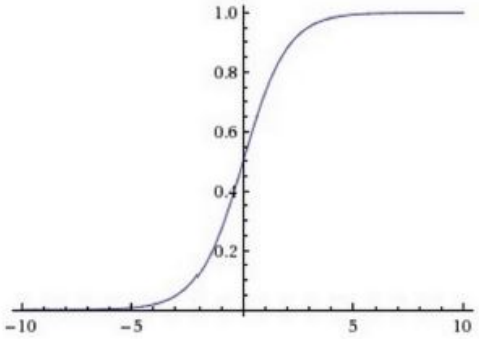


Figura 2.14: Funzione sigmoide.

catena vanno a moltiplicare tutti i gradienti calcolati in precedenza e quelli successivi, conducendo così all'annullamento del gradiente globale.

$$\begin{aligned} \sigma'(z) &= \sigma(z)(1-\sigma(z)) \\ \sigma'(z)_{z\to\infty} &= 0 \\ \sigma'(z)_{z\to-\infty} &= 0 \end{aligned} \tag{2.31}$$

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, come mostrato in Figura 2.15. È 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 'operazione $\exp(\cdot)$ al denominatore è molto costosa dal punto dal punto di vista computazionale, soprattutto rispetto alle alternative che verranno presentate di seguito.

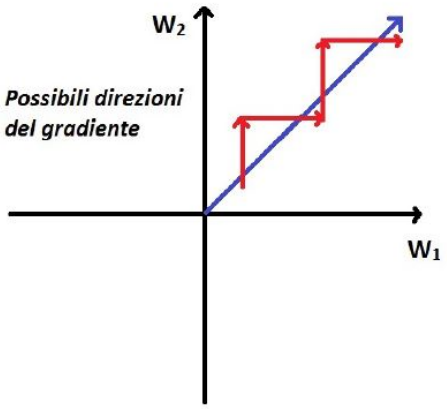


Figura 2.15: Esempio di dinamica a zig-zag nell'aggiornamento di due pesi W1 e W2. La freccia blu indica indica l'ipotetico vettore ottimale per la discesa del gradiente, mentre le frecce rosse i passi di aggiornamento compiuti: gradienti tutti dello stesso segno comportano due sole possibili direzioni di aggiornamento.

2.4.2 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.

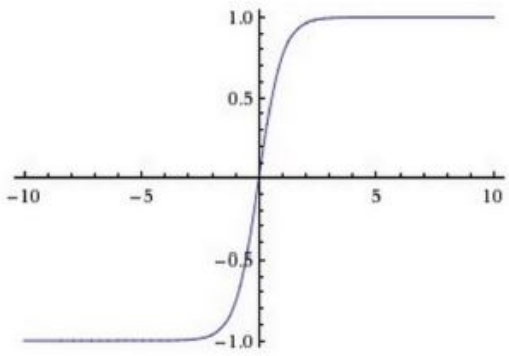


Figura 2.16: Andamento tangente iperbolica.

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2.32}$$

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

2.4.3 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.

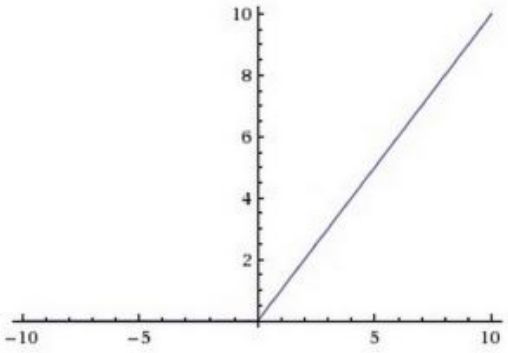


Figura 2.17: Andamento ReLU.

$$ReLU(z) = max(0, z) \tag{2.33}$$

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 (2.34).

$$\frac{ReLU(z)}{\partial z} = \begin{cases} 1 \text{ if } z > 0 \\ 0 \text{ if } z < 0 \end{cases} \tag{2.34}$$

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 : 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 affoondino nella "zona morta".

2.4.4 Leaky ReLU

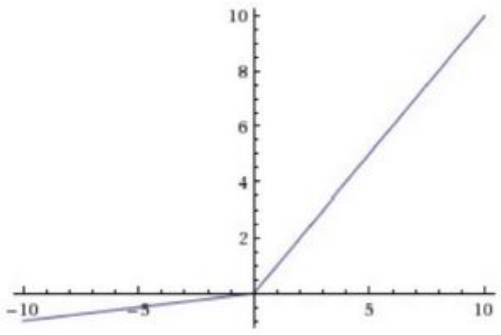


Figura 2.18: Andamento Leaky ReLU.

$$LeakyReLU(z) = max(\alpha z, z) \tag{2.35}$$

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, e 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 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.

2.5 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 a 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 $z^{(k)}$:

$$\begin{aligned} Var(z_l^{(k)}) &= Var\left(\sum_{j=1}^{p_{k-1}} \left(w_{lj}^{(k-1)} a_j^{(k-1)}\right)\right) \\ &= \sum_{j=1}^{p_{k-1}} \Big(E[w_{lj}^{(k-1)}]^2 Var(a_j^{(k-1)}) + E[a_j^{(k-1)}]^2 Var(w_{lj}^{(k-1)}) \\ &+ Var(w_{lj}^{(k-1)}) Var(a_j^{(k-1)}) \Big) \end{aligned} \tag{2.36}$$

Se si assume che le attivazioni siano distribuite simmetricamente intorno allo zero e ci sia indipendenza con i pesi, la (2.36) diventa :

$$\begin{aligned} Var(z_l^{(k)}) &= \sum_{j=1}^{p_{k-1}} Var\left(w_{lj}^{(k-1)}\right) Var\left(a_j^{(k-1)}\right) \\ &= p_{k-1} Var\left(w_{lj}^{(k-1)}\right) \end{aligned} \tag{2.37}$$

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

$$w_{lj}^{(k-1)} \sim N\left(0, \frac{1}{p_{k-1}}\right) \tag{2.38}$$

La (2.38) 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 (2.36) alla (2.37) 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 (2.38) :

$$w_{lj}^{(k-1)} \sim N\left(0, \frac{2}{p_{k-1}}\right) \tag{2.39}$$

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.

2.5.1 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 $z_{l|j}^{(k)}$ input dello strato k generati dall'*i-esima* osservazione, la procedura ideale prevederebbe di sbiancare congiuntamente gli input lungo l'intero dataset χ. Una procedura di questo tipo risulta però molto costosa dal momento che richiede il calcolo dell'intera matrice di covarianza $Cov[z] = E_{z\in\chi}[zz^T] - E[z]E[z]^T$. 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, $z = (z_1, ..., z_{pk})$, si tratta di normalizzare ogni componente l:

$$\hat{z}_l^{(k)} = \frac{z_l^{(k)} - E[z_l^{(k)}]}{\sqrt{Var[z_l^{(k)}]}} \tag{2.40}$$

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 :

$$t_l^{(k)} = \gamma_l^{(k)} \hat{z}_l^{(k)} + \beta_l^{(k)} \tag{2.41}$$

$\gamma_l^{(k)}$ e $\beta_l^{(k)}$ vengono stimati assieme agli altri parametri del modello, e ripristinano 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, 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 $z_{l|j}^{(k)}$ generato dalle m osservazioni del *batch* corrente:

$$B = \left\{ z_{l|1}^{(k)}, ..., z_{l|m}^{(k)} \right\} \tag{2.42}$$

Algorithm 2 Batch Normalization

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

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

1: $\mu_B \leftarrow \dfrac{1}{m} \sum_{i=1}^{m} z_{l|j}^{(k)}$ (Media del batch)

2: $\sigma_B^2 \leftarrow \dfrac{1}{m} \sum_{i=1}^{m} \left(z_{l|j}^{(k)} - \mu_B \right)^2$ (Varianza del batch)

3: $\hat{z}_{l|j}^{(k)} \leftarrow \dfrac{z_{l|j}^{(k)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ (Normalizzaione)

4: $t_{l|j}^{(k)} \leftarrow \gamma_l^{(k)} \hat{z}_{l|j}^{(k)} + \beta_l^{(k)} \equiv BN_{\gamma_l^{(k)}, \beta_l^{(k)}}(z_{l|j}^{(k)})$ (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 (2.41), 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:

$$\frac{\partial f}{\partial \hat{z}_{l|i}^{(k)}} = \frac{\partial f}{\partial t_{l|i}^{(k)}} \gamma_l^{(k)}$$

$$\frac{\partial f}{\partial \sigma_B^2} = \sum_{i=1}^{m} \frac{\partial f}{\partial \hat{z}_{l|i}^{(k)}} (z_{l|i}^{(k)} - \mu_B)(-\frac{1}{2})(\sigma_B^2 + \epsilon)^{-\frac{3}{2}}$$

$$\frac{\partial f}{\partial \mu_B} = \left(\sum_{i=1}^{m} \frac{\partial f}{\partial \hat{z}_{l|i}^{(k)}} \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} \right) + \frac{\partial f}{\partial \sigma_B^2} \frac{\sum_{i=1}^{m} -2(z_{l|i}^{(k)} - \mu_B)}{m}$$

$$\frac{\partial f}{\partial z_{l|i}^{(k)}} = \frac{\partial f}{\partial \hat{z}_{l|i}^{(k)}} \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial f}{\partial \sigma_B^2} \frac{2(z_{l|i}^{(k)} - \mu_B)}{m} + \frac{\partial f}{\partial \mu_B} \frac{1}{m}$$

$$\frac{\partial f}{\partial \gamma_L^{(k)}} = \sum_{i=1}^{m} \frac{\partial f}{\partial t_{l|i}^{(k)}} \hat{z}_{l|i}^{(k)}$$

$$\frac{\partial f}{\partial \beta_l^{(k)}} = \sum_{i=1}^{m} \frac{\partial f}{\partial t_{l|i}^{(k)}}$$

L'algoritmo 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 ì l'allenamento della rete. Per questo motivo, in fase di test, gli input vengono normalizzati sulla base delle statistiche relative all'intera popolazione:

$$\hat{z} = \frac{z - E[z]}{\sqrt{Var[z] + \epsilon}} \tag{2.43}$$

dove $Var[z] = \frac{m}{m-1} E_B[\sigma_B^2]$, con valore atteso calcolato lungo tutti i batches di dimensione m.

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 3

Convolution Neural Network

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, spettrogrammi, dove la griglia è bidimensionale.
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*.
Il motivo principale che ha spinto i ricercatori nello studiare queste reti neurali è data dall'inadeguatezza della struttura *fully connected*.
Infatti, 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 $32x32x3$ (32 altezza, 32 larghezza, 3 canali colore), un singolo neurone connesso in questa maniera comporterebbe un numero totale di $32x32x3 = 3072$ 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 lato comporterebbe un carico di $256x256x3 = 196.608$ 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.

3.1 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 3.1 aiuta a comprendere meglio questa operazione in realtà molto semplice, attraverso un esempio nel caso di un input bidimensionale.

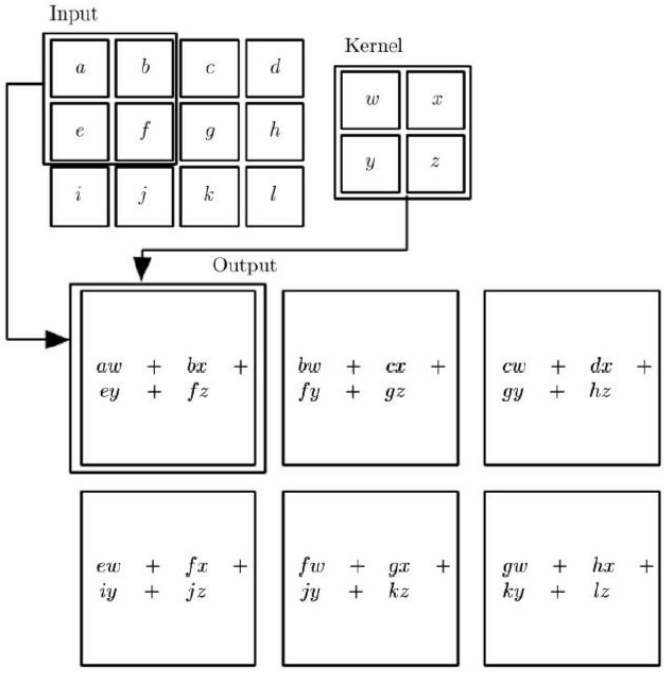


Figura 3.1: Operazione di convoluzione nel caso bidimensionale: un filtro di dimensione $2x2$ viene moltiplicato elemento per elemento per una porzione dell'input di uguali dimensioni.n L'output dell'operazione è costituito dalla somma di tali prodotti. L'operazione di convoluzione viene infine ripetuta spostando il filtro lungo le due dimensioni dell'input.

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 3.2), la quale costituisce di fatto il primo strato nascosto della rete.

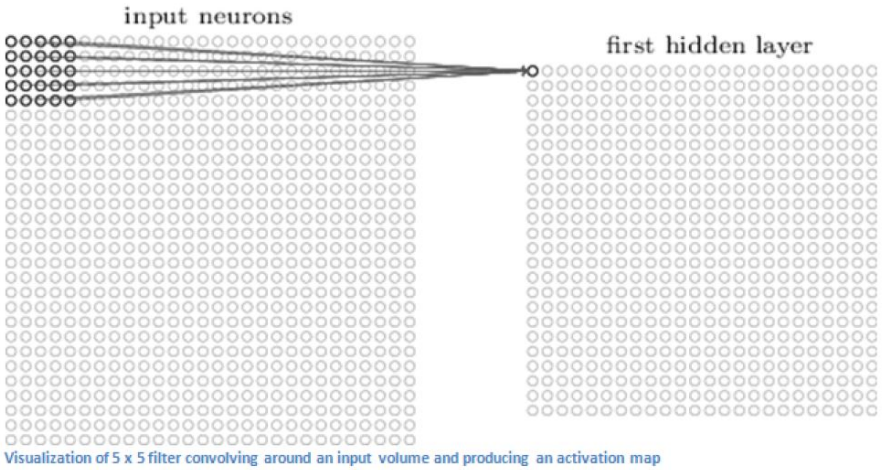


Figura 3.2: Operazione di convoluzione di un filtro di dimensioni $5x5$ e relativa mappa di attivazioni prodotta.

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 2, la transizione dallo strato di input al primo strato nascosto diventa:

$$\begin{aligned} z_{i,j}^{(2)} &= w_0^{(1)} + \sum_{a=1}^{F}\sum_{b=1}^{F} w_{a,b}^{(1)} x_{(i+a-1,j+b-1)} \\ a_{(i,j)}^{(2)} &= g^{(2)}\left(z_{(i,j)}^{(2)}\right) \end{aligned} \tag{3.1}$$

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

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 3.3 aiuta a comprendere meglio questo concetto: viene mostrato

un esempio di convoluzione di due filtri ($W0$ e $W1$) su un input tridimensionale $7x7x3$.

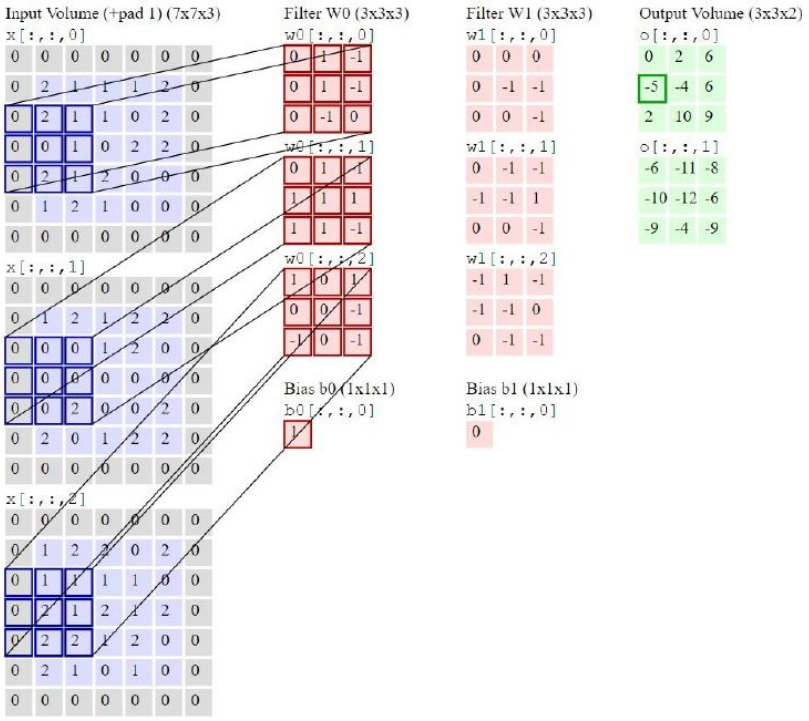


Figura 3.3: Operazione di convoluzione di due differenti filtri ($W0$ e $W1$) di dimensione $3x3x3$ su un volume di input $7x7x3$.

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 (3.1) diventa :

$$\begin{aligned} z^{(2)}_{(i,j)} &= w^{(1)}_0 + \sum_{c=1}^{D} \sum_{a=1} F \sum_{b=1}^{F} w^{(1)}_{(a,b,c)} x_{(i+a-1,j+b-1,c)} \\ a^{(2)}_{(i,j)} &= g^{(2)}\left(z^{(2)}_{(i,j)}\right) \end{aligned} \tag{3.2}$$

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

3.2 Strati di una CNN

3.2.1 Convolution 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 (3.2) introducendo l'indice relativo al filtro (f) :

$$\begin{aligned} z^{(2)}_{f,(i,j)} &= w^{(1)}_0 + \sum_{c=1}^{D} \sum_{ac=1}^{F} \sum_{b=1}^{F} w^{(1)}_{f,(a,b,c)} x_{(i+a-1,j+b-1,c)} \\ a^{(2)}_{f,(j,i)} &= g^{(2)}\left(z^{(2)}_{f,(i,j)}\right) \end{aligned} \tag{3.3}$$

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 zeropadding*, 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 (appartenenti 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:

$$O = \frac{(I - F + 2P)}{S} + 1 \tag{3.4}$$

La profondità coincide invece con il numero di filtri N_F ultilizzati all'interno dello strato. Pertanto, dato in input un volume di dimensioni $W1$ x $H1$ x $D1$ il convolutional layer produrrà un volume di output $W2$ x $H2$ x $D2$ dove :

$$\begin{aligned} W_2 &= \frac{W_1 - F + 2P}{S} + 1 \\ H_2 &= \frac{H_1 - F + 2P}{S} + 1 \\ D_2 &= N_F \end{aligned} \tag{3.5}$$

Chiaramente, i valori di stride e zero-padding devono essere scelti in modo tale che la (3.4) restituisca un valore intero. Ad esempio, l'architettura di *Krizhevsky et al.* (2012) accetta in input immagini di dimensione [227 x 227 x 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 x 55 x 96]. Ognuno dei 55 x 55 x 96 neuroni di questo volume è connesso ad una regione di dimensione [11 x 11 x 3] del volume di input, e tutti i 96 neuroni appartenenti alla stessa colonna profondità sono connessi alla stessa regione [11 x 11 x 3], ma ovviamente attraverso set diverso di pesi.

Weight Sharing Invariance to Translation Nell'esempio sopra riportato, ognuno dei 55 x 55 x 96 = 290.400 neuroni del primo convolutional layer possiede 11 x 11 x 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 x 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 x 11 x 3) x 96 + 96 = 34.944, rispetto ai precedenti 105 milioni, ovvero circa tremila volte meno.

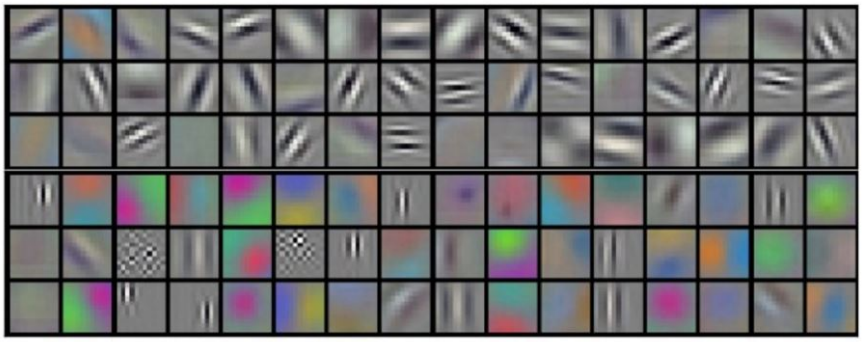


Figura 3.4: Set di 96 filtri 11 x 11 x 3 appresi dall'architettura di Krizhevsky et al. (2012) in un problema di classificazione di immagini. L'assunzione di weight sharing è ragionevole dal momento che individuare una linea o uno spigolo è importante in qualsiasi posizione dell'immagine, e di conseguenza 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.

3.2.2 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 sovradattamento. Il *Pooling Layer* opera indipendentemente su ogni mappa di attivazioni applicando un filtro di dimensione F x 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 $W1$ x $H1$ x $D1$ di input, dai due iperparametri richiesti, ossia

stride (S) ed estensione spaziale del filtro (F):

$$\begin{aligned} W_2 &= \frac{W_1 - F}{S} + 1 \\ H_2 &= \frac{H_1 - F}{F} + 1 \\ D_2 &= D_1 \end{aligned} \tag{3.6}$$

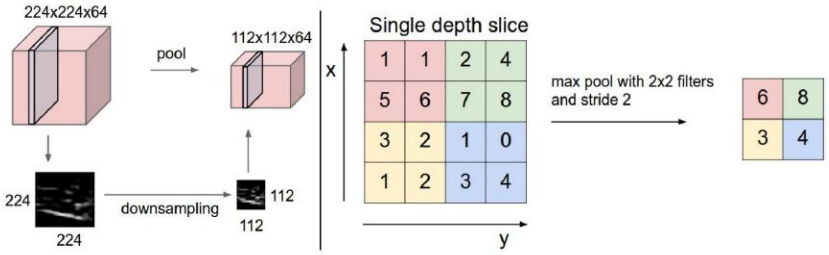


Figura 3.5: Esempio di max-pooling: il filtro opera indipendentemente su ogni features-map e di conseguenza la profondità del volume rimane inalterata. Larghezza e altezza vengono ridimensioante invece di un fattore $F = 2$ (di conseguenza viene eliminato il 75% dei pesi).

Quando il kernel prende il valore massimo parliamo di *Max Pooling Layers*, mentre quando fa la media degli elementi parleremo di *Average Pooling Layers*.

3.2.3 Fully Connected Layers

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*).

3.3 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

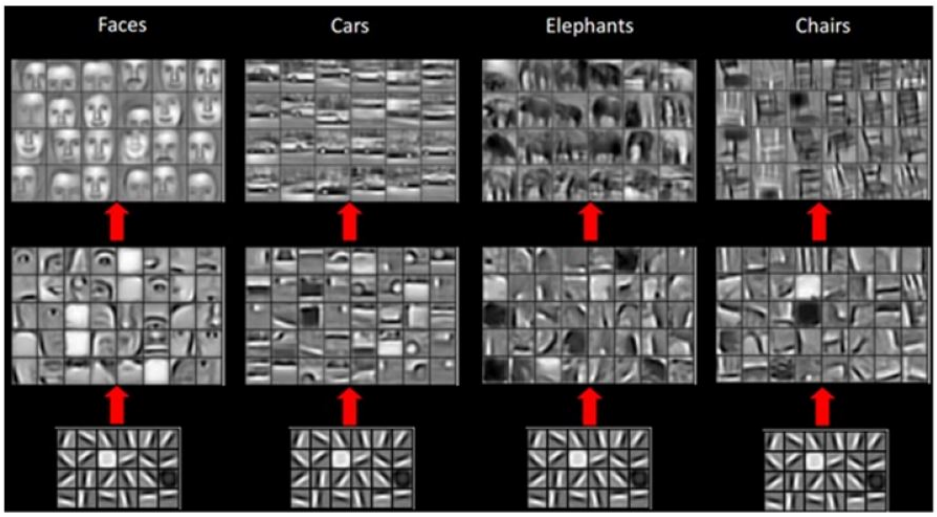


Figura 3.6: Esempio di apprendimento delle features su un dataset di oggetti misti.

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 fiitri 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 x 5 applicata ad immagini 500 x 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 suficiente 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).

3.4 Data augumentation

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 riessione e la rotazione, ma sono da applicare con più prudenza in quanto è necessario prima assicurarsi che non inuiscano sulla corretta classicazione 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 4

Dataset

In questo capitolo andremo a vedere quello che è il dataset da noi utilizzato e come è stato processato per poter essere utilizzato nell'addestramento di una rete neurale. Come vedremo andremo ad utilizzare il dataset ESC.

4.1 ESC

Il dataset è scaricabile attraverso la pagina GitHub del professor *Karol J. Piczak* dell'Università di Varsavia come qui di seguito mostrato :
`https://github.com/karolpiczak/ESC-50`
L'Environmental Sound Classification, che da questo punto in poi chiameremo semplicemente ESC, è un dataset di suoni ambientali formato da 50 tipi di suoni diversi che in totale raggruppano 2000 file audio da 5 secondi ognuni. Queste 50 classi a loro volta possono essere raggruppati in 5 gruppi maggiori, che sono *animali, suoni naturali, suoni umani, suoni domestici, rumori urbani.*

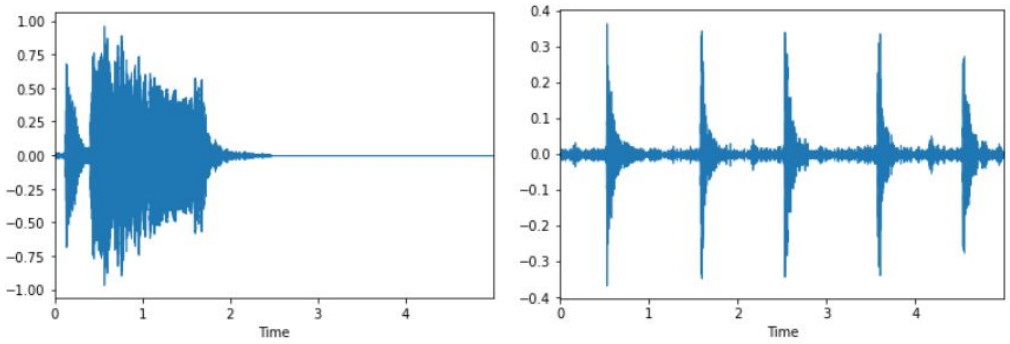
Tabella 4.1: Divisione ESC-50.

Animals	**Natural Soundscapes**	**Human sound**	**Interior** sound	**Urban noise**
Dog	Rain	Crying baby	Door knock	Helicopter
Rooster	Sea waves	Sneezing	Mouse click	Chainsaw
Pig	Crackling fire	Clapping	Keyboard typing	Siren
Cow	Crickets	Breathing	Wood creaks	Car horn
Frog	Chirping birds	Coughing	Can opening	Engine
Cat	Water drops	Footsteps	Washing machine	Train
Hen	Wind	Laughing	Vacuum cleaner	Church bells
Insects	Pouring water	Brushing teeth	Clock alarm	Airplane
Sheep	Toilet flush	Snoring	Clock tick	Fireworks
Crow	Thunderstorm	Drinking snipping	Glass breaking	Hand saw

Come possiamo vedere dalla tabella 4.1, le 50 classi che costituiscono l'intero dataset sono stati divisi nei 5 gruppi maggiori descritti in precedenza.
Sempre dalla tabella possiamo notare che alcuni suoni sono scritti in rosso. Questo perchè le 10 classi colorate compongono quello che viene chiamato ESC-10, ovvero un dataset facente parte dell'ESC, ma con meno classi.

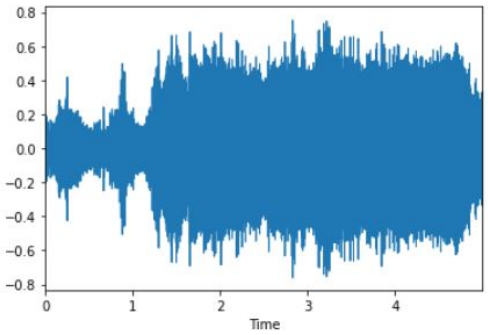
4.2 Pre-processing

Il suono è un fenomeno fisico prodotto dalla vibrazione di un corpo in oscillazione come, negli strumenti musicali, l'aria messa in movimento dal soffio di chi suona un flauto o dalle vibrazioni delle corde di una chitarra. Anche la nostra voce è il prodotto di una vibrazione. Quando parliamo, infatti, l'aria esce dai polmoni e fa vibrare le corde vocali. La vibrazione di un corpo produce una variazione della pressione dell'aria generando delle onde che si propagano fino al nostro apparato uditivo che, trasformate in impulsi, vengono recepite dal cervello come una sensazione sonora.



(a) Rooster

(b) Clock tick.



(c) Chainsaw.

Figura 4.1: Suoni nel tempo

In Figura 4.1 possiamo notare alcuni suoni appartenenti al dataset di nostro interesse. Sull'asse delle ascisse possiamo notare il tempo, mentre sull'asse delle ordinate abbiamo l'intensità del suono. Quest'ultima grandezza, insieme ad altre, fanno parte delle *caratteristiche del suono* che qui di seguito andremo a spiegare.

- **Frequenza :** è il numero di oscillazioni che compie un'onda sonora nell'unità di tempo (t), cioè $f = \frac{n}{t}$. Essa si misura in *hertz (Hz)* che corrisponde ad un'oscillazione completa di un suono in un secondo $\Rightarrow$ 1 Hz = 1 oscillazione al secondo.

- **Intensità :** dipende dall'ampiezza dell'onda sonora. L'intensità si misura in decibel (*dB*) secondo una scala che va da 0 dB (percettibile a stento dall'orecchio), fino a 120 dB (soglia del dolore).
- **Timbro :** cambia a seconda della sorgente da cui proviene ed è quello che differenzia tutti i tipi di suono. I suoni puri sono quelli prodotti dal *diapason*, uno strumento acustico usato per accordare gli strumenti musicali.

Purtroppo i suoni così come gli ascoltiamo non sono utili per addestrare le nostre *Convolution Neural Network*, ma devono essere trasformate in qualcosa di più consono. Ciò significa che dovremo applicare delle operazioni matematiche ed estrarre delle *features*.

4.2.1 Spettrogrammi

Un segnale audio è composto da diverse onde sonore a frequenza singola. Quando vengono campionati segnali audio nel tempo, vengono catturate solo le ampiezza. La ***trasformata di Fourier*** è una formula matematica che ci permette di scomporre un qualunque segnale nelle sue singole frequenze, oltre a darci le rispettive ampiezze. In altre parole, converte il segnale dal dominio del tempo nel *dominio della frequenza*. Il risultato viene chiamato *spettro*.

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft} dt \tag{4.1}$$

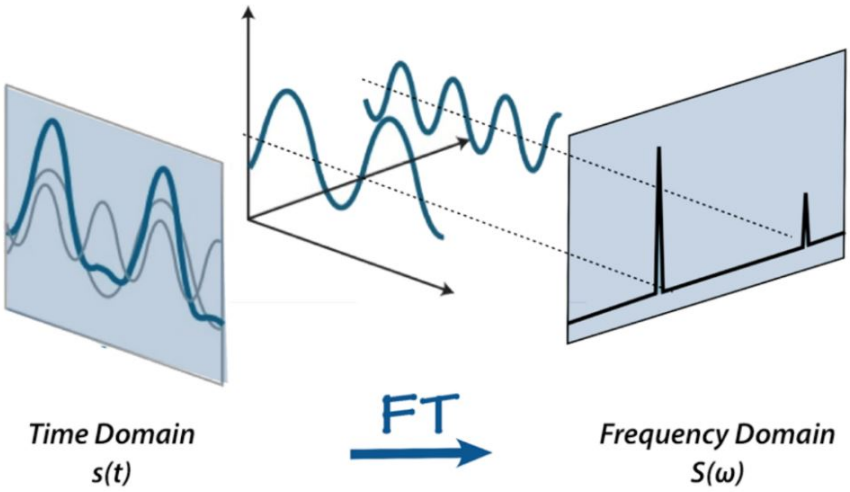
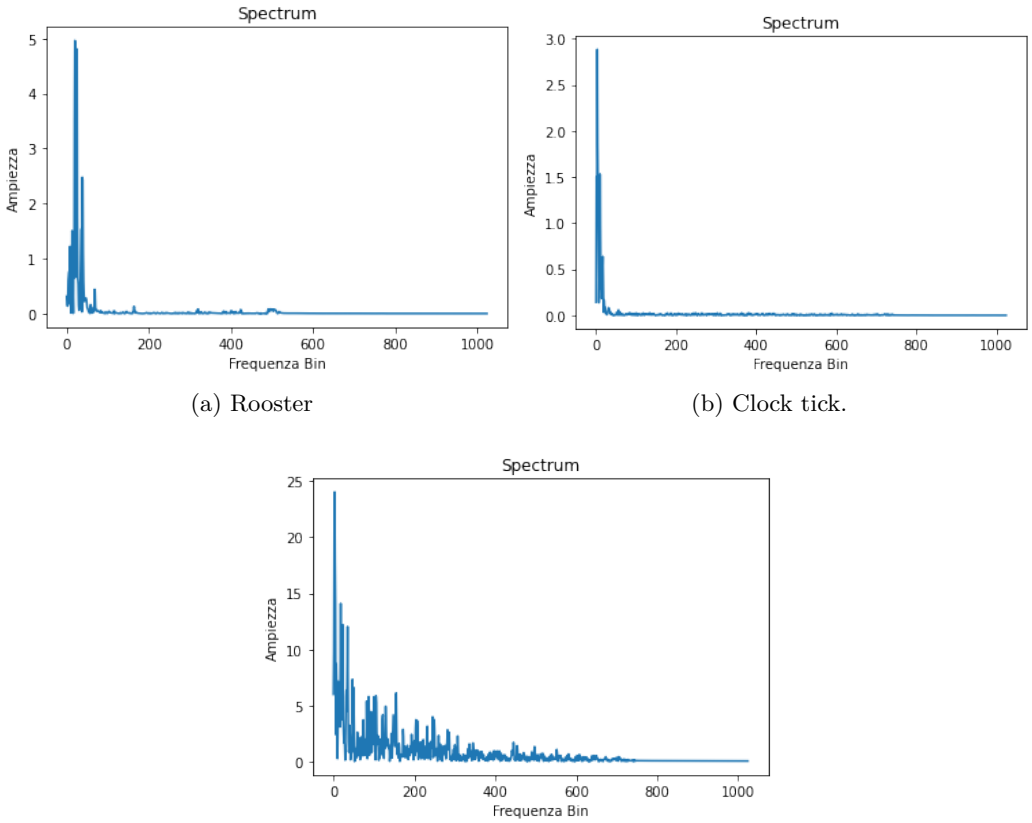


Figura 4.2: Trasformata di Fourier

Ciò è possibile perchè ogni segnale può essere scomposto in un insieme di "*onde seno*" e "*onde coseno*" che se sommate restituiscono il segnale originale.

Dal punto di vista computazionale, la trasformata di Fourier viene calcolata con l'uso della **FFT**, ovvero la *Fast Fourier Transform*. Senza entrare nel merito della trattazione matematica, possiamo dire che la FFT è un algoritmo ottimizzato per calcolare la *trasformata discreta di Fourier* o la sua inversa.



(a) Rooster

(b) Clock tick.

(c) Chainsaw.

Figura 4.3: Suoni nel dominio della frequenza

In Figura 4.3 abbiamo le trasformate di Fourier dei tre suoni di Figura 4.1.
Anche se con la trasformata di Fourier siamo in grado di ricavare delle importanti caratteristiche nel dominio della frequenza, non siamo in grado di sapere cosa succede se il contenuto in frequenza del segnale audio varia nel tempo. Questo è il caso della maggior parte dei segnali audio, noti anche come *segnali non periodici* o *segnali aperiodici*.
Allora quello che viene fatto è calcolare delle FFT su diverse porzioni del segnale d'ingresso con finestre sovrapposte per ottenere quello che viene chiamato **spettrogramma**.

Lo spettrogramma è un modo per rappresentare visivamente l'intensità di un segnale, poichè varia nel tempo a frequenze diverse. Come ultimo passaggio, l'asse y (frequenze) viene convertito in scala logaritmica e la dimensione del colore viene converita in dB. Quest'ultimo passaggio viene fatto perchè gli essere umani sono in

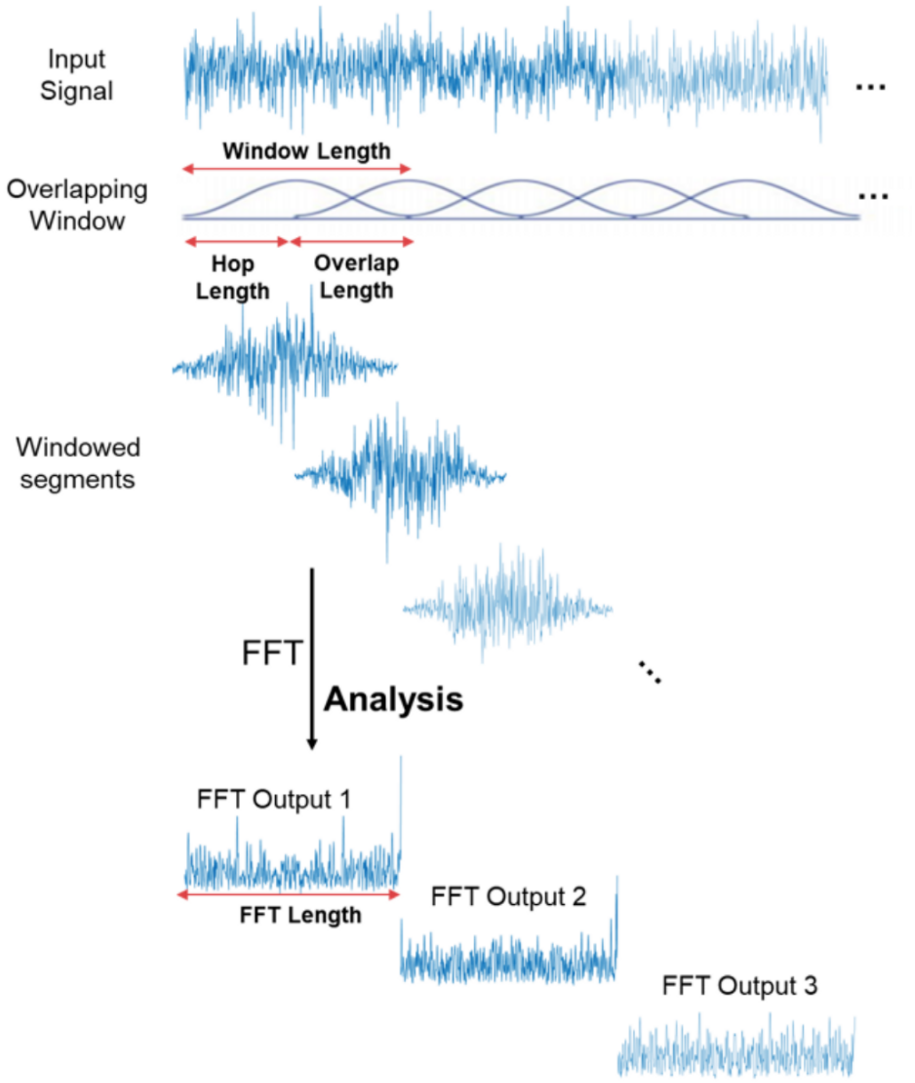


Figura 4.4: Calcolo delle FFT per spettrogramma

grado di percepire solo una gamma molto piccola e concentrata di frequenze e ampiezze.
In Figura 4.4 è rappresentato una schema base su come calcolare uno spettrogramma. Solitamente i valori classici sono :

- Frequenza di campionamento : 44100 Hz
- Lunghezza della finestra : 2048 campioni
- Overlap : 512 campioni

La finestra che viene utilizzata è quella di Hamming :

$$w[n] = 0.54 - 0.46cos\left(\frac{2\pi n}{N-1}\right) \tag{4.2}$$

dove $0 \leq n \leq N-1$ e N è la lunghezza della finestra.

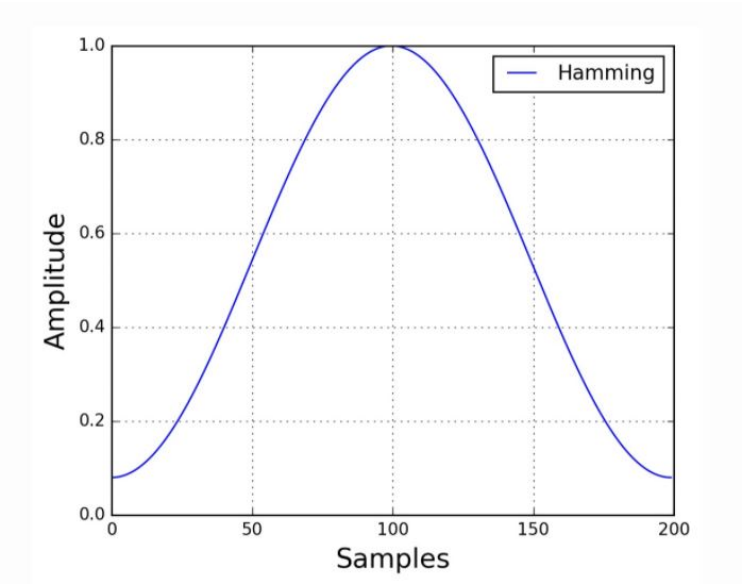


Figura 4.5: Finestra di Hamming

Una volta effettuate queste operazioni otteniamo gli spettrogrammi che sono utili per l'addestramento delle reti neurali.

4.2.2 Mel-spectrogram

Tuttavia, le prestazioni allo stato dell'arte sono state ottenute utilizzando rappresentazioni più discriminanti come *Mel filterbank features* [9], le *Gammatone features* [10] e le *wavelet-based features* [11].
In questo progetto di tesi utilizzeremo gli spettrogrammi convertiti nella *scala Mel*. Gli studi hanno dimostrato che gli esseri umani non percepiscono le frequenze su una scala lineare. Siamo più bravi a rilevare le differenze nelle frequenze più basse rispetto alle frequenze più alte. Ad esempio, possiamo facilmente capire la differenza tra 500 e 1000 Hz, ma difficilmente saremo in grado di distinguere tra 10.000 e 10.500 Hz, anche se la distanza tra le due coppie è la stessa.
Nel 1937, *Stevens*, *Volkmann* e *Newmann* proposero un'unità di intonazione tale che distanze uguali in intonazione suonassero ugualmente distanti dall'ascoltatore. Questa è chiamata **scala Mel**. Eseguiamo un'operazione matematica sulle frequenze per convertirle nella scala Mel.
La scala Mel è una funzione definita a tratti nel seguente modo :

$$mel(f) = \begin{cases} 1 & \text{se } f \leq 1 \text{ kHz} \\ 2595 * log(1 + \frac{f}{700}) & \text{se } f > 1 \text{ kHz} \end{cases} \tag{4.3}$$

Il nome "**mel**" deriva dalla parola *melodia* ad indicare che la scala è basata su confronti di altezze (pitch). In Figura 4.6 vediamo graficamente come avviene la

trasformazione delle frequenze.

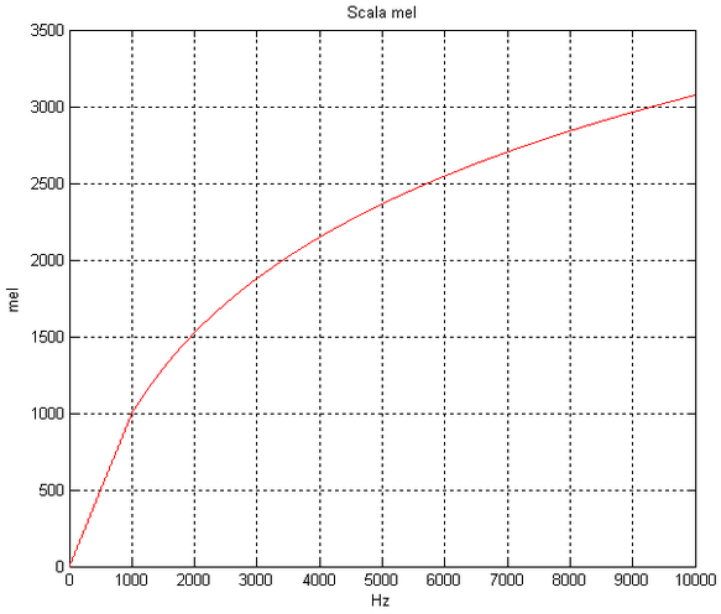
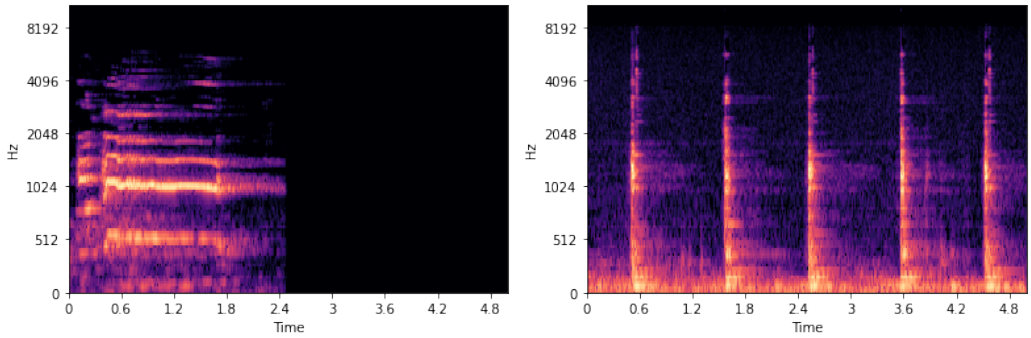


Figura 4.6: Grafico funzione Mel

In conclusione, dopo aver calcolato la trasformata di Fourier, fatto lo spettrogramma e trasformato la scala delle frequenze in scala Mel otteniamo degli spettrogrammi come in Figura 4.7

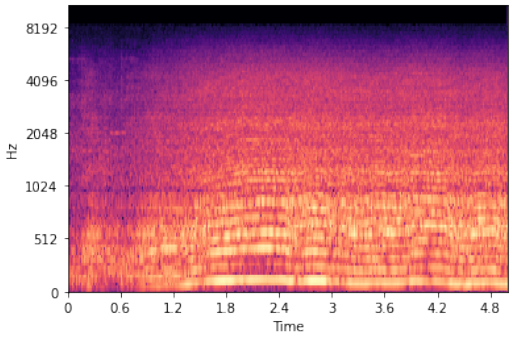
4.2.3 Divisione spettrogrammi

Naturalmente creando uno spettrogramma per ogni file audio siamo in grado di generare un numero di 40 spettrogrammi per classe, non sufficiente per addestrare una rete neurale. Infatti, facendo riferimento al dataset *"Mnist"* abbiamo che ogni classe ha esattamente 10.000 immagini.
Inoltre c'è da pensare che dovendo fare un dimostratore live bisognerebbe attendere la registrazione di un qualunque suono per un tempo pari a 5 secondi, impensabile per una qualunque applicazione real time.
Allora l'idea è quella di prendere ogni spettrogramma e dividerlo in blocchi, in modo che abbia una durata inferiore ai 5 secondi. A questo punto il problema è scegliere la giusta finestra temporale per avere un buon compromesso tra lo shape dell'input (in modo da addestrare meno parametri all'interno della rete), un minor tempo occupato dalla registrazione in fase di dimostrazione, ma anche avere una buona accuracy.
Zhichao Zhang *et al.* [12] ha proposto una CNN dove i file audio vengono campionati a 44.1 kHz, e i relativi spettrogrammi sono calcolati con un numero di filtri Mel pari a 128. Da questi ricavano, splittando, degli spettrogrammi che presentano una matrice uguale a $128x128$, quindi un tempo di registrazione circa pari a 1,5 s.



(a) Rooster

(b) Clock tick.



(c) Chainsaw.

Figura 4.7: Spettrogrammi degli audio di Figura 4.1

Diversamente da [12], Piczak [13] ha anch'esso diviso gli spettrogrammi in blocchi più piccoli, ma campionando il segnale a 22050 Hz. Inoltre vengono usati 60 filtri Mel e lo spettrogramma viene splittato in modo che la matrice abbia dimensione $60x41$, facendo si che ogni registrazione sia di circa 950 ms.
In questo lavoro di tesi sono state svolte numerose prove riguardanti quale sia la migliore finestra temporale per avere il giusto compromesso. Inizialmente si è partiti con l'avere delle matrici di dimensione $128x8$, che è circa 100 ms. Questa finestra temporale però non ha avuto un buon funzionamento perchè molto ridotta. Infatti, suoni come il verso del *cane* hanno bisogno di maggior tempo per essere del tutto espletate.
Successivamente si è pensato di utilizzare una matrice $128x128$ come descritto da Zhang, ma questo comportava delle dimensioni esagerate per alcuni microcontrollari che hanno poca memoria di RAM, oltre al fatto che la registrazione è di circa 1,5 s. Il giusto compromesso tra durata e buona accuratezza si è raggiunta quindi con una finestra temporale pari a circa 800 ms, che presupponeva una matrice di dimensione $128x64$. Questo tempo è stato ritenuto ottimo sia per quei suoni che hanno una durata più prolungata rispetto ad altri, ma anche per suoni che possono essere periodici come il *clock tick*.
In definitiva ogni "piccolo spettrogramma" è stato ottenuto finestrando una porzione

dello spettrogramma originario, e tra due spettrogrammi adiacenti c'è una sovrapposizione del 50%.
Per completare la parte di pre-processing manca solo un ultimo passaggio, e forse il più fondamentale. Infatti nel dataset creato da Piczak ci sono dei file audio che nell'arco dei 5 secondi non presentano alcun suono, ma sono dei semplici momenti di *silenzio* (parte finale di Figura 4.1a). Essendo il nostro un *addestramento supervisionato*, potrebbe esserci il rischio che ad uno spettrogramma che rappresenta proprio questi momenti di silenzio venga assegnata la label della classe dello spettrogramma originale, causando problemi durante l'addestramento.
Ogni articolo presenta delle tecniche diverse per eliminare i momenti di silenzio. La maggior parte di questi articoli agiscono direttamente sui file audio andando ad effettuare l'operazione di "*silence trimming*", ovvero eliminare la parte di audio dove l'ampiezza del suono è nulla (silenzio) e lasciando invariata la parte "buona".
In questa tesi, la procedura di rimozione è stata un'altra, prendendo spunto dall'articolo di Wang [14]. Wang, lavorando sul "*Continual Learning*", usa degli spettrogrammi che hanno una dimensione pari a $128x16$ eliminando gli spettrogrammi che hanno una *norma di Frobenius* maggiore di $1e^{-4}$. La *norma matriciale* è una funzione che associa ad una matrice uno scalare ($||\cdot|| : K^{m\times n} \to \mathbb{R}^+$) e presenta delle proprietà :

- $||\mathbb{A}|| = 0$ se e solo se $\mathbb{A} = 0$ (matrice nulla)
- $||\lambda\mathbb{A}|| = |\lambda|||\mathbb{A}||$
- $||\mathbb{A} + \mathbb{B}|| \leq ||\mathbb{A}|| + ||\mathbb{B}||$

In questo caso, la *norma di Frobenius* è la seguente :

$$||X||_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2} \tag{4.4}$$

Nel nostro caso (data la matrice di dimensione maggiori) non si può pensare di utilizzare in questo modo la norma, perchè uno spettrogramma sostanzialmente "vuoto" potrebbe avere una minima parte di suono utile che porterebbe la norma matriciale ad essere sopra il valore di soglia. L'idea è di calcolare la norma di Frobenius sulle colonne della matrice e vedere nel vettore risultante quanti zeri ci sono che corrispondono al silenzio. La (4.4) diventa :

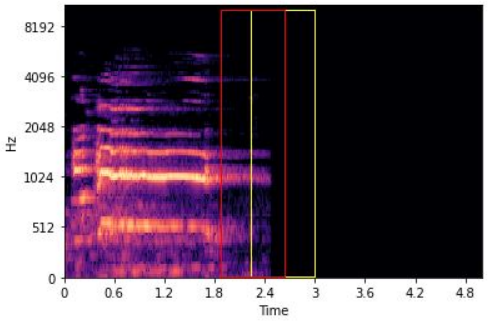
$$||X||_F = |x_1 \dots x_n| \tag{4.5}$$

dove

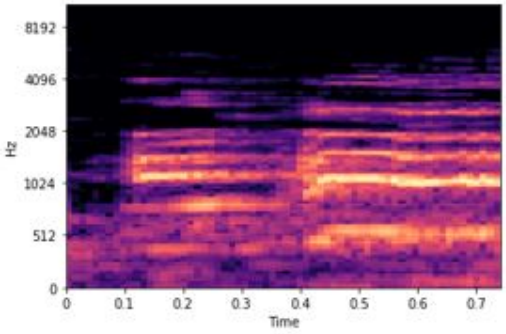
$$x_j = \sqrt{\sum_{i=1}^{m}|aij|^2} \tag{4.6}$$

se quante vettore è composto da zeri almeno nel 75% della sua lunghezza, allora verrà scartato e considerato silenzio.

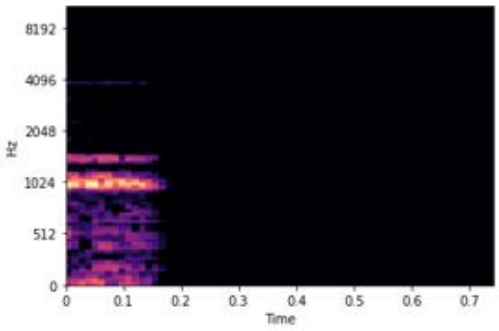
In Figura 4.8a vediamo un esempio di come viene finestrato uno spettrogramma; in Figura 4.8b abbiamo uno spettrogramma che presenta del suono utile e quindi viene salvato per allenare la rete, mentre in Figura 4.8c abbiamo uno spettrogramma che presenta per la maggior parte del tempo del silenzio e quindi viene scartato.



(a) Finestratura spettrogramma (5s)



(b) Spettrogramma salvato (750ms)



(c) Spettrogramma eliminato (750ms)

Figura 4.8: Finestratura spettrogramma

Questa è la procedura che viene seguita per ogni file audio di ogni classe per poter ottenere il giusto dataset che deve andare ad allenare la rete.
Per completezza riportiamo l'algoritmo 3 con i passaggi fondamentali.

4.3 Data augumentation

Una volta terminata la fase di creazione del dataset potremmo iniziare a parlare della rete sviluppata e dei relativi risultati, ma c'è ancora una cosa da fare. Dovendo costruire un dimostratore live, sicuramente andando a registrare degli audio non avremo mai dei suoni puliti come lo sono quelli del dataset di Piczak. Proprio per questo motivo si ha la necessità di dover fare del *data augumentation*. Il nostro esperimento applica cinque diverse trasformazione al dataset, che poi vengono salvate in file .npy separatamente. La libreria python "*Librosa*" [15] viene usata per effettuare queste trasformazioni.

Algorithm 3 Creazione dataset

input : File audio

1: Campionamento del file audio con $f_c = 44100Hz$
2: Calcolo dello spettrogramma con :

- window length : 2048 samples
- hop size : 512 samples
- Mel filter : 125

3: Finestratura dello spettrogramma con relativo salvataggio o eliminazione degli spettrogrammi ricavati.

Qui di seguito le cinque trasformazioni applicate con i relativi fattori :

- **Shift positivo del pitch : +2**
- **Shift negativo del pitch : -2**
- **Stretch time rapido : 1,2**
- **Stretch time lento : 0,7**
- **Aggiunta del rumore bianco**

Tutte queste trasformazioni sono utili per poter allenare la nostra rete su un set di dati più ampio, in modo che sia in grado di riconoscere tutte le possibili sfaccettature di ogni singolo audio.
In tabella 4.2 vediamo i dettagli del dataset ESC-10 prima e dopo il data augumentation.

Dataset	**Numero di file audio**	**Dataset in GB**	**Durata (min)**
ESC-10	400	168	33
ESC-10(with augumentation)	2400	1008	231

Tabella 4.2: Dettagli ESC-10

Capitolo 5

Analisi della rete neurale proposta

5.1 Convolution Neural Network

Una volta che è stata descritta tutta la parte inerente alla parte di pre-processing e di come viene ricavato l'intero dataset, siamo pronti per poter parlare della rete neurale usata.
La rete che verrà presentata di seguito si isprime a delle reti neurali già esistenti per questo tipo di lavoro, dato che comunque il lavoro finale è basato sull'addestramento di una rete "tiny" che deve essere in grado di classificare dei file audio mai visti su un micro controllore.
La rete è ispirata dall'articolo di Zhang [12] e può essere vista come la concatenazione di tre strati principali, dove ogni strato è formato dalla cascata di due *Convolution layer* e un *Max-pooling layer.*

Layer	**Numero filtri**	**Dimesione filtro**	**Stride**	**Regolarizzazione**
Convolution	8	3x3	1x1	l2 = 0.0001
Batch Normalization				
Convolution	8	3x3	1x1	l2 = 0.0001
Batch Normalization				
Max-Pooling		2x2	2x2	
Convolution	16	4x1	1x1	l2 = 0.0001
Convolution	16	4x1	1x1	l2 = 0.0001
Max-Pooling		2x2	2x2	
Convolution	32	1x4	1x1	l2 = 0.0001
Dropout	$\phi = 0,2$			
Convolution	32	2x4	1x1	l2 = 0.0001
Dropout	$\phi = 0,3$			
Max-Pooling		1x3	2x2	
Flatten	1792			
Dropout	$\phi = 0,4$			
Dense	128			
Dropout	$\phi = 0,5$			
Dense	10			

Tabella 5.1: Architettura rete neurale

La tabella 5.1 è la struttura sequenziale della CNN usata in questo lavoro di tesi. Come già detto, è una versione modificata di Zhang per poter essere adattata al nostro *input shape* che è $128x64x1$ a differenza del $128x128x2$ usato da Zhang.
In questo caso usiamo i primi due layer convolutivi come un estrattore base delle features. In successione usiamo il secondo strato con un kernel $4x1$ per apprendere i pattern sull'asse della frequenza, mentre il terzo strato è esattamente l'opposto e viene usato per apprendere dei pattern sull'asse temporale.
Come si può vedere, nel primo strato abbiamo due layer di *Batch normalization* subito dopo i layer convolutivi. Questo perchè ci permette di rendere questa rete più veloce e più stabile attraverso la normalizzazione del livello di input mediante ricentramento e ridimensionamento. Viene effettuata anche la regolarizzazione dei layer convolutivi e verso la parte finale della rete abbiamo aggiunto dei layer di *Dropout* per evitare il fenomeno dell'overfitting; ovvero imparare a memoria i dati di train e non saper riconoscere dei dati che non ha mai visto.
Della rete possiamo dire che ha un *input shape* pari a $128x64$ e prima di entrare nel layer *Flatten* la dimensione dello spettrogramma si è ridotta fino a diventare $14x4x32$ dove 32 è il numero di filtri applicati dall'ultimo strato convolutivo.
Possiamo vedere che con l'avvento delle reti convolutive siamo riusciti ad abbassare il numero di neuroni del primo strato di una rete neurale classica. Infatti si è passati a dover avere 8.192 neuroni ad un numero pari a 1.792, nettamente minore. Questa rete, come vedremo nei paragrafi successivi, è stata progettata per rientrare nel limite dei 250.000 parametri (quindi essere sotto il MegaByte di occupazione Flash).
Come preannunciato questa rete presenta per l'esatezza un numero di parametri pari a 243.394, di cui 243,362 sono quelli trainabili, e i restanti 32 sono i parametri non trainabili del *Batch normalization*.
Inoltre, bisogna dire per completezza che tutti i layer usano come attivazione la funzione non lineare "*relu*", mentre l'ultimo layer *Dense* usa la funzione non lineare "*softmax*" per le motivazioni descritte in 2.4.

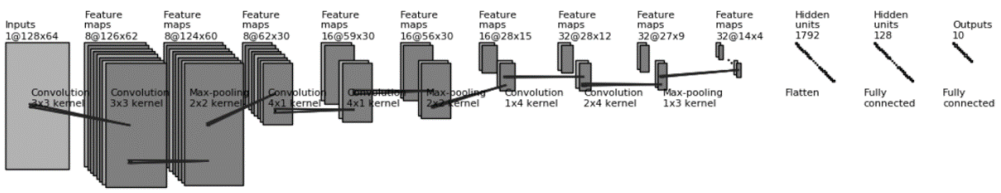


Figura 5.1: Convolution Neural Network proposta.

5.2 Iperparametri

Una piccola sezione di questo capitolo verrà spesa per descrivere quelli che sono stati gli *iperparametri* scelti per l'addestramento della rete neurale :

- **Bach size = 50**. Il batch size definisce il numero di dati che attraversano la rete neurale prima che i parametri interni vengano aggiornati [16]. Alla fine di ogni batch, i risultati predetti vengono confrontati con quelli reali e viene calcolato l'errore.
- **Epoche = 300**. Il numero di epoche definisce quante volte l'intero dataset va ad attraversare la rete neurale andando ad aggiornare i parametri. Questo valore consente l'esecuzione dell'algoritmo di apprendimento finché l'errore del modello non è stato sufficientemente ridotto al minimo. È necessario trovare un compromesso tra questo numero e l'effettiva minimizzazione dell'errore e del tempo di calcolo [17].
- **Learning rate** = $1e^{-3}$. Come spiegato in 2.3.1 il learning rate è il parametro più importante per l'addestramento di una rete neurale. Infatti valori troppo alti tendono ad addestrare subito la rete rischiando di girare attorno ad un minimo locale. Valori troppo bassi hanno una convergenza più lenta che rischia di bloccarsi in un punto di minimo locale non ottimale senza riuscire ad uscire. Per questo motivo si è pesanto di adottare un *learning rate variabile*, ovvero in base al numero di epoche andare a variare quello che è il learning rate. Inizialmente si ha un valore più alto e col passare delle epoche tende a scendere seguendo la forma di un gradino.

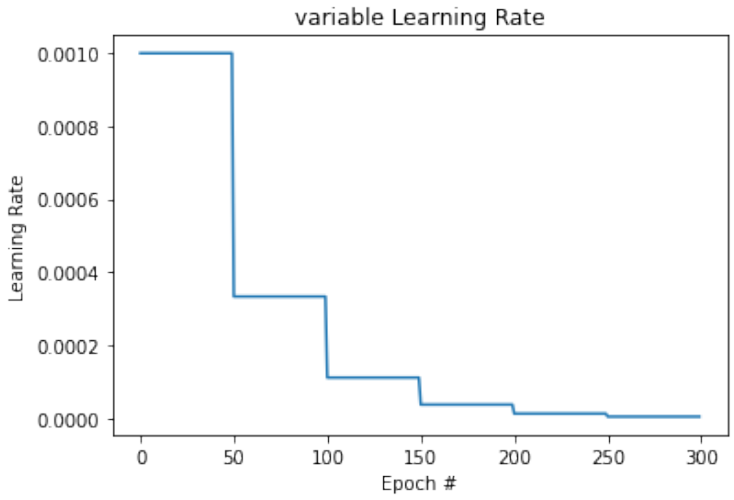


Figura 5.2: Learning rate variabile.

- **Loss function = Categorical Crossentropy**.
- **Ottimizzatore = SGD**.

- **Pesi e bias = inizializzazione randomica**.

5.3 K-fold validation

La **K-fold validation** è una tecnica statistica utilizzabile in presenza di una buona numerosità del campione osservato. In particolare, la convalida incrociata cosiddetta k-fold consiste nella suddivisione dell'insieme di dati totale in k parti di uguale numerosità e, a ogni passo, la k^a parte del'insieme di dati viene a essere quella di convalida, mentre la restante parte costituisce sempre l'insieme di addestramento. Così si allena il modello per ognuna delle k parti, evitando quindi problemi di sovradattamento, ma anche di campionamento asimmetrico (e quindi affetto da distorsione) del campione osservato, tipico della suddivisione dei dati in due sole parti (ossia addestramento/convalida). In altre parole, si suddivide il campione osservato in gruppi di egual numerosità, si esclude iterativamente un gruppo alla volta e si cerca di predirlo coi gruppi non esclusi, al fine di verificare la bontà del modello di predizione utilizzato.

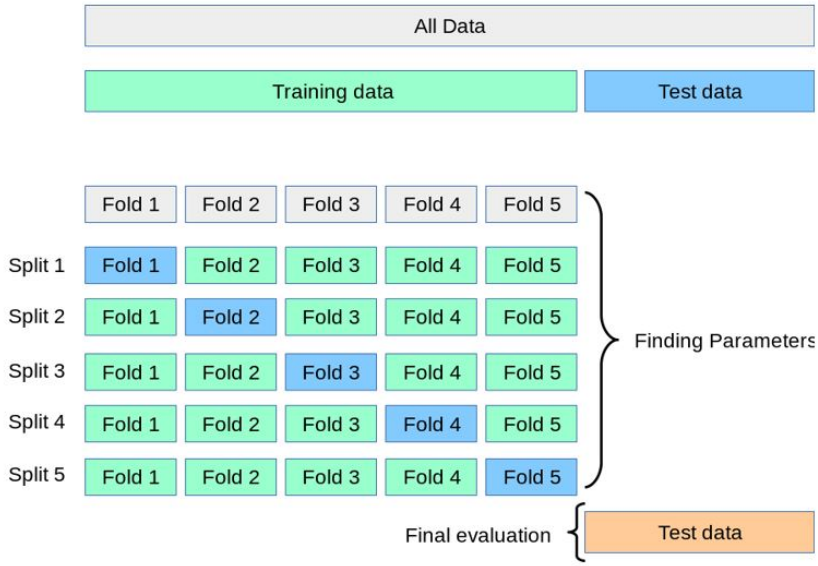


Figura 5.3: Divisione del dataset per K-fold validation

La convalida incrociata è usata anche perchè generalmente si traduce in una stima meno distorta o meno ottimistica dell'abilità del modello rispetto ad altri metodi, come una semplice suddivisione di addestramento / prova.

5.4 Risultati training

Dopo aver addestrato la rete attraverso la k-fold validation andiamo a discutere i risultati che si sono ottenuti confrontandoli anche con le reti di altri articoli.

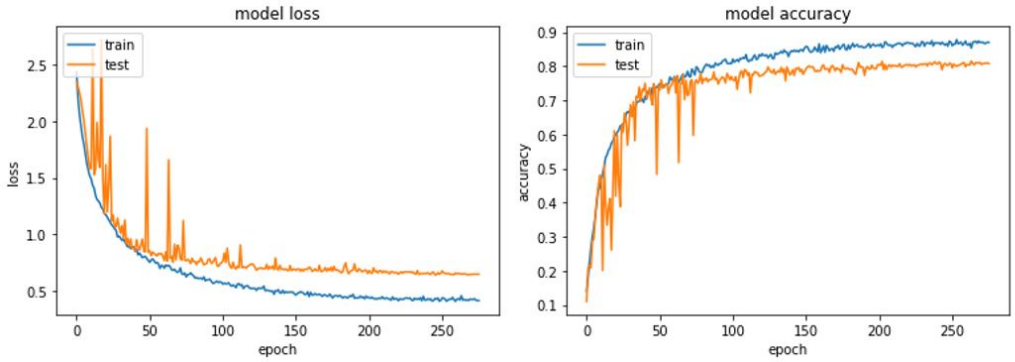


Figura 5.4: Loss ed accuracy del training

Dalla Figura 5.4 vediamo come le curve arancioni del validation test siano leggermente peggiori rispetto alle blu del train test. Questo è un risultato che ci si aspetta dall'addestramento di una rete neurale che viene appunto addestrata sul set di train.
Una cosa interessante che possiamo notare è che nelle prime 50 epoche, quando il learning rate è abbastanza alto, la rete raggiunge con una pendenza elevata il valore di 70% di accuracy, per poi aumentare più gradualmente.
Nonostante la rete sia "piccola" in numero di parametri, comunque riusciamo a raggiungere un'accuracy intorno all'80%.
Nella tabella seguente confrontiamo i risultati di questa tesi con quelli di Zhang [12] e di Piczak [13].

Modello	Numero parametri	Accuracy(%)
Modello proposto	243.398	79,8%
Piczak		80,5%
Zhang	1.446.258	88,7%

Tabella 5.2: Confronto risultati

Come possiamo notare Zhang riesce ad ottenere un'accuracy nettamente migliore rispetto a quella proposta in questo lavoro, ma allo stesso tempo notiamo che questa rete ha quasi sei volte il nostro numero di parametri, quindi ho un maggior numero di filtri per l'estrazione delle features.
Di seguito (Figura 5.5) invece riportiamo la Confusion Matrix per vedere come vengono scambiati le classi. Come possiamo vedere, la maggior parte delle classi sono conprese tra il 75% e l'85%, che significa che la rete ha imparato abbastanza uniformemente tutte le classi che compongono il dataset.

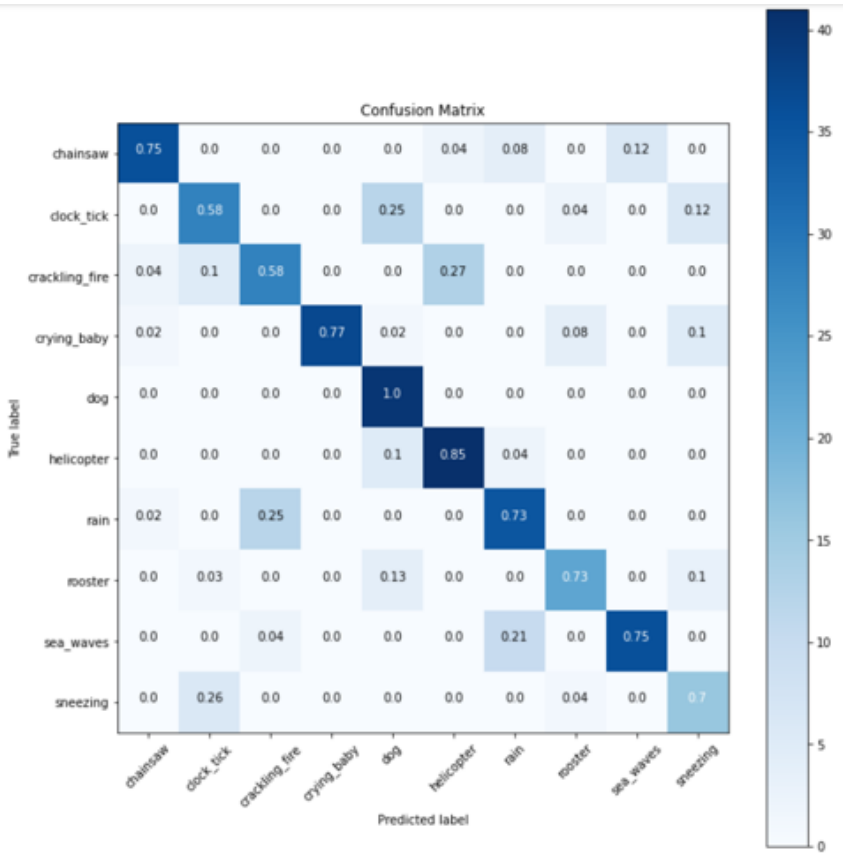


Figura 5.5: Confusion matrix

5.5 Post-training quantization

Prima di andare a vedere quelli che sono i risultati finali, è bene parlare di un'ultima procedura che è stata effettuata per cercare di abbassare ulteriormente la quantità di memoria necessaria per poter classificare una rete neurale su un microcontrollore. *X-Cube AI* (tool di ST che si occupa di intelligenza artificiale) può essere usato anche per generare un modello quantizzato.
La quantizzazione è una tecnica di ottimizzazione per distribuire un modello *float32* riducendo la dimensione (quindi comporterà uno spazio di archiviazione più piccolo e un minor utilizzo della memoria in fase di esecuzione), migliorando la latenza della CPU/MCU a discapito di una leggera degradazione dell'accuracy.
Ci sono due possibili forme di quantizzazione : la *post-training quantization* e la *quantization aware training*. La differenza tra le due è che la prima è più facile da usare perchè ha solo bisogno di una rete pre-addestrata, mentre la seconda è leggermente più complessa perchè la quantizzazione è effettuata nello stesso tempo in cui c'è l'addestramento della rete, anche se questo porta a una minore degradazione dell'accuracy finale.
Al momento X-Cube AI è in grado di fornire la quantizzazione solo per le reti in Keras e in Tensorflow.

Nel nostro caso si è utilizzato la *post-training quantization*, dove lo schema generale è riportato in Figura 5.6.

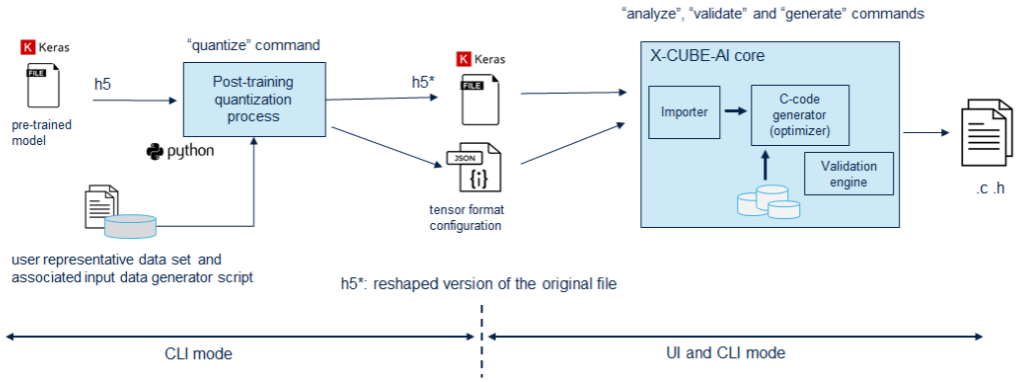


Figura 5.6: Processo post-training quantization

La prima parte della procedura è tutta effettuata nella *Command Line Interface* (che verà abbreviato con CLI) della ST, dove abbiamo bisogno della rete pre-addestrata in formato .h5 e una piccola parte di dati che servono al processo per conoscere alcuni parametri e calcolare l'accuracy finale rispetto all'accuracy del modello non quantizzato.
Terminata la procedura ci vengono restituiti due file :

- **model.h5*** che è semplicemente una versione rimodellata di quella originale;
- **.json file** che contiene la configurazione dei tensori quantizzati;

La seconda parte che può essere eseguita sia sul programma che in CLI è la classica procedura di X-Cube AI per la validazione del modello sulle board STM32 (che vedremo più avanti).
Avendo detto che il processo di post-training quantization è effettuato attraverso CLI bisogna settare un file json dove andare ad inserire tutte le informazioni necessarie.

```
{
    "model_name": "ESC10",
    "path_to_floatingpoint_h5": ".\\checkpoints\\128x64_augumentation_09_01_74.36%.h5",
    "quant_test_set_dir": "",
    "quant_test_ratio": "0.8",
    "evaluation_test_set_dir": "",
    "batch_size": "128",
    "modules_directory": ".\\ESC10\\",
    "filename_test_set_generation": "test_set_generation.py",
    "filename_quantizer_algos": "quantizer_algos_user.py",
    "algorithm": "MinMax",
    "arithmetic": "Integer",
    "weights_integer_scheme": "UnsignedAsymmetric",
    "activations_integer_scheme": "UnsignedAsymmetric",
    "output_directory": ".\\ESC10\\",
    "per_channel": "False"
}
```

Figura 5.7: Setting post-training quantization

Come vediamo in Figura 5.7 bisogna settari vari parametri, dove andremo a spiegare quelli principali :

- **filename_test_set_generation** : è uno script Python per il caricamento del dataset di test;
- **algorithm** : deve essere inserito quale algoritmo di quantizzazione si vuole usare. Nel nostro caso si è scelto il "*MinMax*" che è un processo basato sul minimo e massimo di tutti i tensori per pesi e attivazioni. Il set di test di quantizzazione viene utilizzato per stimare gli intervalli di attivazione. I pesi sono costanti e quindi i loro intervalli vengono stimati direttamente.
- **arithmetic** : indica l'aritmetica che viene utilizzata.
- **integer scheme** : si sceglie se usare l'aritmetica simmetrica o asimmetrica. Nel nostro caso si è scelta l'aritmetica intera asimmetrica.

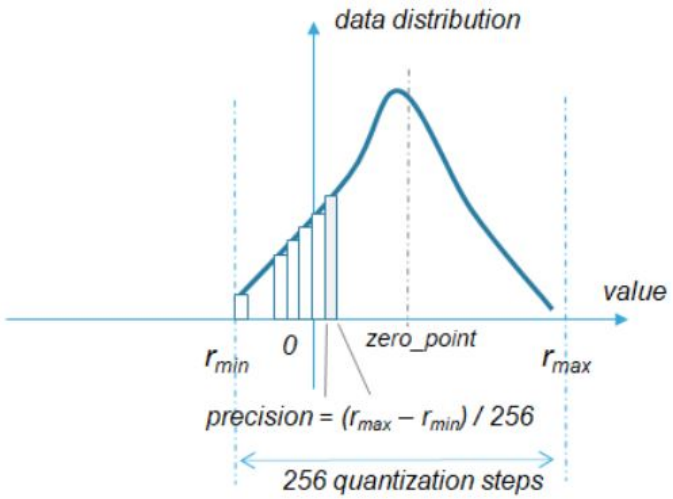


Figura 5.8: Integer quantization (int8/asymmetric)

Ciò significa che il valore quantizzato è calcolato come

$$q = \frac{r}{scale} + zero\_point \tag{5.1}$$

dove r è il valore float32, $zero\_point$ può essere un qualsiasi valore nell'intervallo [-128,127] (signed 8b) o nell'intervallo [0,255] (unsigned 8b).

5.6 X-Cube AI

Arrivati a questo punto non rimane altro che andare a verificare se i risultati ottenuti attraverso l'addestramento su Colab, saranno confermati anche sulle board della ST. Il modo per farlo è attraverso l'uso di X-Cube AI.

X-CUBE-AI è un pacchetto di espansione STM32Cube che fa parte dell'ecosistema STM32Cube.AI e che estende le capacità di STM32CubeMX con la conversione automatica della rete neurale pre-addestrata e l'integrazione della libreria ottimizzata generata nel progetto dell'utente.

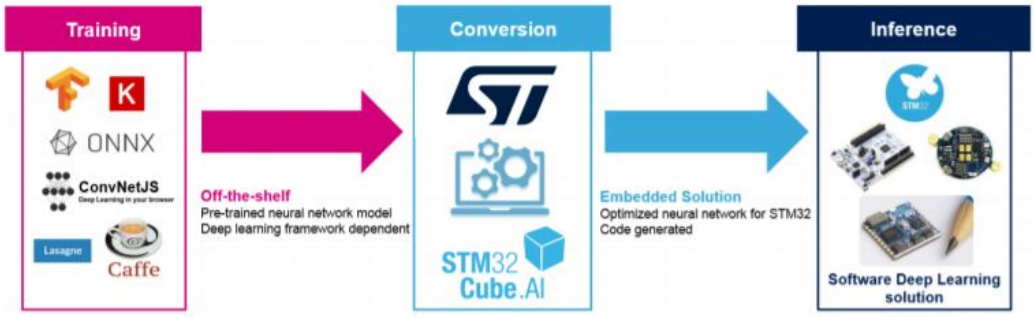


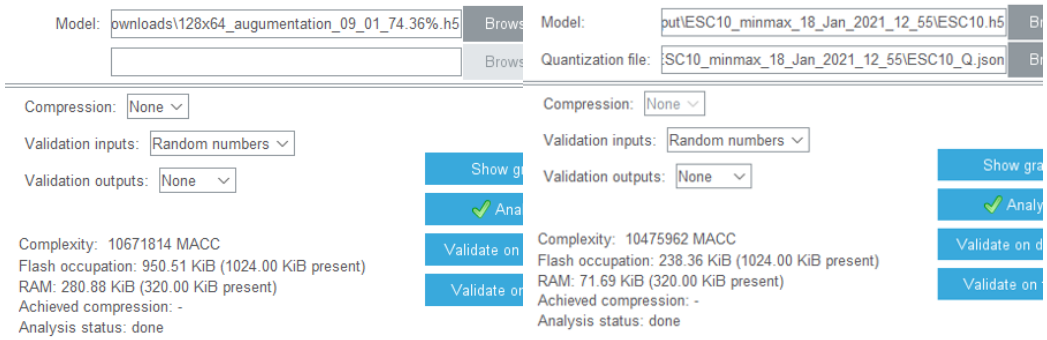
Figura 5.9: X-Cube AI overwiev

Il pacchetto di espansione X-CUBE-AI offre anche diversi mezzi per convalidare i modelli di rete neurale sia su PC desktop che su STM32, oltre a misurare le prestazioni su dispositivi STM32 senza il bisogno di scrivere del codice C da parte dell'utente.

Model	Parametri	Flash (KB)	RAM (KiB)	MACC
ESC-10 (no aug)	243.394	950,57	278,07	10.671.814
ESC-10 (with aug)	243.394	950,51	278,07	10.671.814
ESC-10 (quantized)	243.394	238,36	71,69	10.475.962

Tabella 5.3: Parametri delle CNN'

Come possiamo vedere dall tabella 5.3 la rete utilizzata è sempre la stessa (descritta nella sezione 5.1), dove le uniche differenze le possiamo apprezzare nella rete quantizzata sotto le voci di "*Flash*" e "*RAM*" i cui valori sono circa quattro volte più piccoli rispetto alla rete non quantizzata.
In Figura 5.10 possiamo vedere come la versione grafica di X-Cube AI ci restituisce i dati inseriti in tabella 5.3 attraverso il comando Analyze.
L'altro parametro fondamentale è il MACC (Multiply-ACCumulated operations) che definisce la complessità computazionale del modello.
Come vedremo nella prossima tabella, questi risultati andranno a trovare una chiara dimostrazione in quello che è il tempo di inferenza (tempo necessario per il passaggio del dato attraverso la rete con la relativa estrazione della features).
Quello che balza subito all'occhio dalla tabella 5.4 è il tempo di inferenza della rete quantizzata. Infatti passando da *float32* a *uint8*, oltre a diminuire notevolmente la size della rete, diminuisce di quasi tre volte il tempo di inferenza. Questo è un ottimo risultato, soprattutto quando parliamo di classificare dei dati con l'utilizzo di microcontrollori, limitati nelle loro prestazioni.



(a) Report ESC-10 (b) Report ESC-10 quantizzato

Figura 5.10: Report X-Cube AI

Model	**Validate on desktop**		**Validate on target**	
	Accuracy	Inference time (ms)	Accuracy	Inference time (ms)
ESC-10 (no aug)	72,49%	41,632	72,49%	1642,410
ESC-10 (with aug)	74,36%	40,416	74,36%	1654,515
ESC-10 (quantized)			74,36%	690,188

Tabella 5.4: Accuracy e inference time della CNN

Naturalmente il tempo di inferenza oltre a dipendere dalla rete, dipende anche dalla frequenza di clock del dispositivo che si sta utilizzando. Quando parliamo di "*Validate on desktop*" significa che stiamo calcolando delle inferenze sul PC dove è installato il software STM32Cube. Per ottenere un tempo medio di $41ms$ abbiamo usato un PC che ha le seguenti caratteristiche :

- Intel Core i7-6700HQ 2,60 GHz
- RAM 8 GB

Un PC che avrà una CPU più performante con un clock più veloce, sicuramente farà in modo di abbassare questo tempo. Lo stesso discorso deve essere fatto anche per la board della ST utilizzata. In questo caso è stata usata una STM32L496G che possiede :

- *Cortex*® M-4 80 MHz
- Flash 1024 KiB
- RAM 320 KiB

Un micro con prestazioni migliori porta ad avere più memoria in cui si può inserire una rete neurale con più parametri, ed un clock più veloce ne ridurrà sicuramente il tempo d'inferenza.

Facendo riferimento alla rete fin qui discussa, possiamo dire che la Nucleo STM32H7, che arriva fino a 480 MHz è in grado di calcolare l'inferenza 6 volte più velocemente rispetto alla board usata da noi. Ciò significa che in caso di rete non quantizzata, il tempo di inferenza sarebbe stato di circa 300ms, mentre per la rete quantizzata saremmo scesi a circa 100ms (il chè ci porta ad avvicinarci alle prestazioni di un PC). Nel prossimo capitolo andremo a discutere di una demo, non totalmente indipendente dal PC, per la classificazione di suoni ambientali.

Capitolo 6

Dimostratore

Come preannunciato, questo capitolo andrà a descrivere una demo creata per la classificazione di suoni ambientali attraverso l'uso di microcontrollori.
Essendo ai primi stadi dello sviluppo, la demo non è interamente sviluppata su microcontrollore, ma una buona percentuale del programma gira su PC. Infatti, il PC ha il compito di registrare attraverso il microfono dei suoni, calcolarne lo spettrogramma e successivamente inviare il dato attraverso la porta seriale. Arrivato il dato al microcontrollore, verrà eseguita l'inferenza e una volta terminata, il vettore delle probabilità di ogni classe verrà restituita al PC che stamperà a video il risultato.
Fatta questa panoramica generale sul programma, andremo a spiegare ogni passo in maniera più dettagliata, per poi andare a fare delle considerazioni finali.

- **Registrazione audio**
  Come detto, la registrazione del file audio viene eseguita con l'utilizzo di un PC. Lo script, scritto interamente in Python, attraverso la libreria open source *sounddevice* è in grado di interagire col microfono interno e registrare per il tempo desiderato. Un difetto di questa libreria è che i primi 350 ms del suono registrato vengono completamente silenziati. Un modo per risolvere è allungare il tempo di registrazione fino a 1100 ms, così che i primi 350 ms vengono scartati e si prendono gli ultimi 750 ms che sono utili per la nostra demo. L'audio è campionato ad un frequenza di 44100 Hz.

- **Calcolo spettrogramma**
  Il secondo passo è calcolare lo spettrogramma del suono registrato. Anche questa operazione è eseguita con l'uso della libreria open source *Librosa*. Lo spettrogramma viene calcolato nello stesso modo descritto nel capitolo 4, ottenendo una matrice di dimensione $128x64$. Per compatibilità con il programma X-Cube AI, allo spettrogramma viene effettuato un reshape della matrice, in modo da avere il dato scritto su un vettore monodimensionale.

- **Invio spettrogramma**
  Una delle parti più complesse e più onerose è stato proprio l'invio dello spettrogramma al micro. Questa operazione è stata eseguita con l'uso di

un'API di ST. Quest'API, scritta sia in C che in Python, serve proprio ad inviare dei dati float ai microcontrollori.
Il dato viene incapsulato in un pacchetto, insieme ad altri dati come il tipo di dato, lo shape e naturalmente lo "start" e l'"end byte".

- **Ricezione dati**
  Essendo l'API scritta con entrambi i linguaggi, la parte di ricezione è esattamente complementare a quella dell'invio. Sul microcontrollare viene riservato in memoria uno spazio inerente alle dimensioni del dato da ricevere, così che la libreria andrà a scrivere il dato ricevuto in questo spazio. A questo punto si è pronti ad effettuare la classificazione.

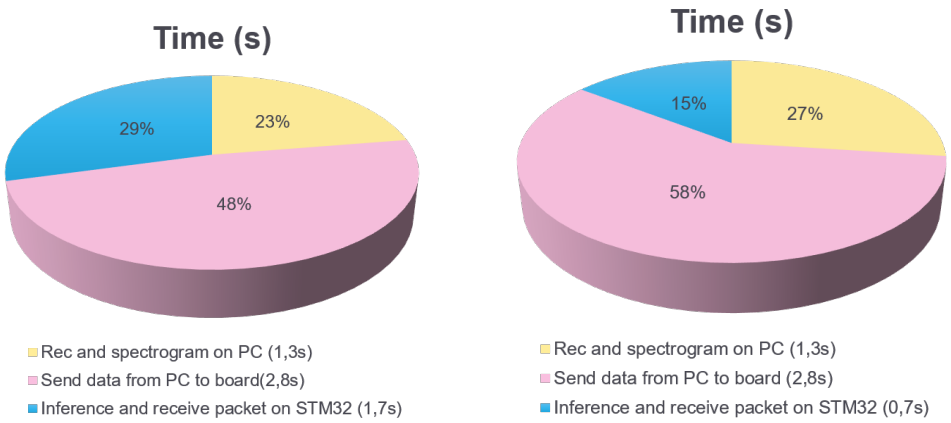
- **Inferenza su micro controllore**
  Questa parte di codice in C serve per poter calcolare l'inferenza dello spettrogramma attraverso la rete pre-addestrata caricata sul micro attraverso X-Cube AI.
  Il pacchetto di intelligenza artificiale della ST è in grado di caricare tutti i pesi e tutte le attivazioni della rete, e con qualche riga di codice si è in grado di immettere lo spettrogramma attraverso gli strati convolutivi della rete, come se stessimo facendo l'addestramento.

- **Classificazione**
  Questo è l'ultimo passaggio della demo qui descritta. All'uscita della rete avremo un vettore di lunghezza pari al numero delle classi, dove ogni numero è la probabilità che lo spettrogramma appartenga a quella classe. Il vettore viene inviato al PC attraverso l'API e con una semplice funzione di *argmax* sapremo a quale classe è stato associato il nostro audio.

Quelli descritti qui sopra sono i passi fondamentali che ci hanno permesso lo sviluppo di questa demo.
In Figura 6.1 possiamo vedere, attraverso questi grafici a torta, come varia la configurazione temporale quando si classifica il dato con la rete quantizzata o no. Per entrambe le reti si ha che il tempo per la registrazione, calcolo dello spettrogramma e invio del dato è uguale perchè non dipende nè dalla rete nè dal microcontrollore. Potremmo giustificare i tempi in questo modo:

- **Parte gialla**. Comprende il tempo necessario per la registrazione e il calcolo dello spettrogramma. Impiega 1,3 s per il difetto della libreria *sounddevice* descritto in precedenza.

- **Parte rosa**. Corrisponde all'invio dello spettrogramma, che è la parte più onerosa della demo impiegando 2,8 s. Infatti è stato scelto l'UART come protocollo di comunicazione e per non avere errori siamo costretti ad usare come baud rate 115.200 bps (bit per second). Potremmo giustificare questo tempo andando a calcolare quanti bit vengono inviati :



(a) Rete non quantizzata (b) Rete quantizzata

Figura 6.1: Tempo per classificazione di un audio

$$128 \times 64 \times 4[byte] \times 8\left[\frac{bit}{byte}\right] = 262.144[bit]$$
$$\Longrightarrow \frac{262.144[bit]}{115.200\left[\frac{bit}{s}\right]} = 2,27[s]$$

Da queste due semplici formule vediamo che per inviare effettivamente il dato ci vogliono circa 2,3 s. Il restante mezzo secondo serve per inviare altri byte del pacchetto dell'API e un pò di tempo sicuramente è dovuto al tempo di print di Python, che sappiamo non essere istantaneo.

- **Parte azzurra**. Tempo dovuto al lavoro del microcontrollore. In questo caso possiamo vedere come il tipo di rete (quantizzata o no) influisce sul tempo d'inferenza. Guardando sempre al caso della STM32L496G vediamo come le due reti abbiamo tempi diversi e quindi si può vedere che effettivamente la rete quantizzata ha prodotto l'effetto voluto, ovvero velocizzare il tempo d'inferenza.

Non potendo integrare un video, andremo a scrivere tutte le classi appartenenti al dataset e se queste vengono classificate correttamente. Inoltre scriveremo i link Youtube usati per la classificazione dei vari suoni.

- **Chainsaw** → OK (`https://www.youtube.com/watch?v=vHE17nh-ZHc`)
- **Clock tick** → OK (`https://www.youtube.com/watch?v=u_jz6lJoG-c`)
- **Crackling fire** → NO (`https://www.youtube.com/watch?v=UgHKb_7884o`)

- **Crying baby** → OK (`https://www.youtube.com/watch?v=6Lp-h1S3rMk`)
- **Dog** → OK (`https://www.youtube.com/watch?v=r32XEPtkY6o`)
- **Helicopter** → OK (`https://www.youtube.com/watch?v=0fm0oU8FO0U`)
- **Rain** → OK (`https://www.youtube.com/watch?v=q76bMs-NwRk`)
- **Rooster** → OK (`https://www.youtube.com/watch?v=K3CenN7d30Y`)
- **Sea waves** → NO (`https://www.youtube.com/watch?v=btmjDyff6E8`)
- **Sneezing** → OK (`https://www.youtube.com/watch?v=FXJlxzeVgSw`)

Le classi scritte in rosso sono quelle che non vengono riconosciute dalla rete neurale. In questo caso la classe "*crackling fire*" viene scambiata solitamente con "*rain*". La classe "*sea waves*" invece con "*chainsaw*" o "*helicopter*". Un'ultima considerazione va fatta sulle classi "*chainsaw*" e "*helicopter*"; infatti essendo molto simili come rumori, solitamente vengono scambiati o comunque le due probabilità sono molto simili.

Capitolo 7

Conclusioni e sviluppi futuri

La sfida all'origine di questa tesi è stata quella di riuscire a costruire una *Tiny CNN* in grado di poter essere eseguita all'interno delle poche risorse di cui dispone un microcontrollore.
Naturalmente, non ci si è concentrati in maniera approfondita sull'avere un'accuracy delle rete elevata per due motivi fondamentali. Il primo è perchè lo scopo futuro è far girare delle reti neurali su micro, quindi si è limitati nel numero dei parametri. In secondo luogo, il dataset ESC non è così ricco di dati tale da permettere un ottimo addestramento della rete.
Nonostante ciò, i risultati ottenuti sia dall'addestramento sia dal dimostratore sono rilevanti e sono una buona partenza per quello che potrebbe essere uno sviluppo futuro.
Lo sviluppo potrebbe seguire due vie differenti, per poi andarsi a ricollegare nuovamente. In primis, si potrebbero cercare nuovi dati e nello stesso tempo migliorare la rete per avere un'accuracy sempre migliore (magari anche con altre strutture che non sono CNN).
Il secondo passo invece è un pò più pratico, quindi sviluppare delle applicazioni con l'uso della rete sviluppata in questa tesi con quelle dieci classi. Un'idea potrebbe essere la *cancellazione attiva del rumore*. Per esempio, ipotizzando che ci sia una chiamata tra due persone, potremmo pensare che ci sia del rumore di sottofondo che disturbi la conversazione. Un micro all'interno del telefonino potrebbe riconoscere il rumore e attuare una modifica per cercare di eliminarlo.

Bibliografia

[1] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. Ieee, 2013.

[2] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing magazine*, 29(6):82–97, 2012.

[3] Michael A Casey, Remco Veltkamp, Masataka Goto, Marc Leman, Christophe Rhodes, and Malcolm Slaney. Content-based music information retrieval: Current directions and future challenges. *Proceedings of the IEEE*, 96(4):668–696, 2008.

[4] Rainer Typke, Frans Wiering, and Remco C Veltkamp. A survey of music information retrieval systems. In *Proc. 6th international conference on music information retrieval*, pages 153–160. Queen Mary, University of London, 2005.

[5] Regunathan Radhakrishnan, Ajay Divakaran, and A Smaragdis. Audio analysis for surveillance applications. In *IEEE Workshop on Applications of Signal Processing to Audio and Acoustics, 2005.*, pages 158–161. IEEE, 2005.

[6] Michel Vacher, Jean-François Serignat, and Stephane Chaillol. Sound classification in a smart room environment: an approach using gmm and hmm methods. In *The 4th IEEE Conference on Speech Technology and Human-Computer Dialogue (SpeD 2007), Publishing House of the Romanian Academy (Bucharest)*, volume 1, pages 135–146, 2007.

[7] Daniele Barchiesi, Dimitrios Giannoulis, Dan Stowell, and Mark D Plumbley. Acoustic scene classification: Classifying environments from the sounds they produce. *IEEE Signal Processing Magazine*, 32(3):16–34, 2015.

[8] Richard F Lyon. Machine hearing: An emerging field [exploratory dsp]. *IEEE signal processing magazine*, 27(5):131–139, 2010.

[9] S. Chu, S Narayanan, and C.C.J. Kuo. Environmental sound recognition with time–frequency audio features. *IEEE Transactions on Audio, Speech, and Language Processing*, 17(6):1142–1158, 2009.

[10] X. Valero and F. Alias. Gammatone cepstral coefficients: Biologically inspired features for non-speech audio classification. *IEEE Transactions on Multimedia*, 14(6):1684–1689, 2012.

[11] J.T. Geiger and K. Helwani. Improving event detection for audio surveillance using gabor filterbank features. In *2015 23rd European Signal Processing Conference (EUSIPCO)*, pages 714–718. IEEE, 2015.

[12] Z. Zhang, S. Xu, S. Cao, and S. Zhang. Deep convolutional neural network with mixup for environmental sound classification. In *Chinese Conference on Pattern Recognition and Computer Vision (PRCV)*, pages 356–367. Springer, 2018.

[13] Karol J Piczak. Environmental sound classification with convolutional neural networks. In *2015 IEEE 25th International Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6. IEEE, 2015.

[14] Z. Wang, C. Subakan, et al. Continual learning of new sound classes using generative replay. In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 308–312. IEEE, 2019.

[15] Brian McFee, Colin Raffel, Dawen Liang, Daniel PW Ellis, Matt McVicar, Eric Battenberg, and Oriol Nieto. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, volume 8, pages 18–25, 2015.

[16] Bruce Schmeiser. Batch size effects in the analysis of simulation output. *Operations Research*, 30(3):556–568, 1982.

[17] Rohit Rawat, Jignesh K Patel, and Michael T Manry. Minimizing validation error with respect to network size and number of training epochs. In *The 2013 international joint conference on neural networks (IJCNN)*, pages 1–7. IEEE, 2013.