



UNIVERSITÀ POLITECNICA DELLE  
MARCHE

DIGITAL ADAPTIVE CIRCUITS AND LEARNING  
SYSTEMS

# Sound Event Detection con la tecnica del “*few-shot learning*”

*Matteo Orlandini e Jacopo Pagliuca*

Prof. Stefano SQUARTINI  
Dott.ssa Michela CANTARINI

20 luglio 2021

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Convolutional Neural Network (CNN)</b>	<b>4</b>
<b>3</b>	<b>Few-shot learning</b>	<b>5</b>
3.1	Introduzione . . . . .	5
3.2	Struttura del meta learning . . . . .	5
<b>4</b>	<b>Prototypical Network</b>	<b>7</b>
<b>5</b>	<b>Relation Network</b>	<b>9</b>
<b>6</b>	<b>Preprocessing del dataset</b>	<b>11</b>
6.1	Dataset Spoken Wikipedia Corpora . . . . .	11
6.2	Le annotazioni del dataset . . . . .	11
6.3	Lettori del dataset . . . . .	13
6.4	Parole degli audio . . . . .	15
6.5	Associazione delle parole con il rispettivo lettore . . . . .	18
6.6	Calcolo dello spettrogramma delle parole . . . . .	19
6.7	Creazione delle feature . . . . .	21
<b>7</b>	<b>Implementazione della Prototypical network</b>	<b>30</b>
7.1	Definizione della rete . . . . .	30
7.2	Training . . . . .	31
7.3	Validation . . . . .	37
7.4	Test . . . . .	37
<b>8</b>	<b>Implementazione della Relation network</b>	<b>41</b>
8.1	Definizione della rete . . . . .	41
8.2	Training . . . . .	43
8.3	Validation . . . . .	48
8.4	Test . . . . .	48
<b>9</b>	<b>Risultati</b>	<b>52</b>
9.1	Area under precision-recall curve (AUPRC) . . . . .	52
9.2	Confronto con l'AUPRC di riferimento . . . . .	54
<b>10</b>	<b>Conclusioni</b>	<b>56</b>

# 1 Introduzione

L'argomento centrale della tesina è l'individuazione di eventi sonori percettivamente simili all'interno di una registrazione. Per fare ciò, sono state implementate due reti neurali presentate nel paper "*Few-Shot Event Detection*" [6].

Varie applicazioni di queste reti possono essere la rilevazione di particolari suoni nella musica o la rimozione di parole di riempimento nei podcast, come ad esempio gli "ehm". Solitamente questo processo viene eseguito manualmente, risultando in un compito difficile e tedioso.

I classici modelli di deep-learning per la determinazione di suoni richiedono una grande mole di dati per fare il training. In questo modo la rete è capace di riconoscere parole sulle quali si è allenata.

Nell'approccio proposto, chiamato *few-shot*, invece, è necessario un piccolo dataset di riferimento. La rete non sarà allenata a riconoscere i suoni che ha ascoltato nel training, ma assumerà la capacità di riconoscere la somiglianza fra due suoni che sta analizzando.

Le reti *few-shot* di solito sono state utilizzate per un set chiuso, dove in un *task* di classificazione veniva presentata una query e confrontata con  $K$  istanze di  $C$  classi, con  $C$  fisso. Questo metodo è chiamato  $C$ -way  $K$ -shot.

Il paper di riferimento invece, cerca di applicare una rete *few-shot* in un contesto più ampio, in cui è necessario riconoscere una parola non vista in precedenza in una sequenza di suoni non classificati precedentemente dalla rete.

In figura 1 è mostrato il metodo proposto, in cui il *few-shot* viene applicato ad un insieme aperto. Inoltre, viene mostrato come costruire un set di esempi negativi e come aumentare i dati positivi per incrementare la precisione senza bisogno di ulteriore sforzo umano nell'etichettatura dei dati.

Prima di fare questo però, la rete deve "*imparare ad imparare*": verrà quindi sottoposta a una serie di task classici di *few shot* nei quali la rete deve riconoscere a quale classe appartiene una parola avendo come riferimento  $C$  classi di parole ciascuna composta da  $K$  istanze.

In ogni episodio di training della rete, le parole da riconoscere sono diverse, generalizzando il più possibile l'evento e allenando la capacità della rete di *riconoscere la somiglianza* tra le parole, ma non le parole stesse.

I modelli di apprendimento *few-shot* sono spesso allenati per risolvere il compito di classificazione  $C$ -way  $K$ -shot, dove  $C$  è il numero fisso di classi tra cui discriminare e  $K$  è il numero di esempi forniti per classe. La figura 2 è un esempio di un task di classificazione 2-way 5-shot, in cui ogni colore rappresenta una classe. Dato un support set di  $C \times K$  esempi etichettati, l'obiettivo del modello è incorporare i set di supporto e query in uno spazio

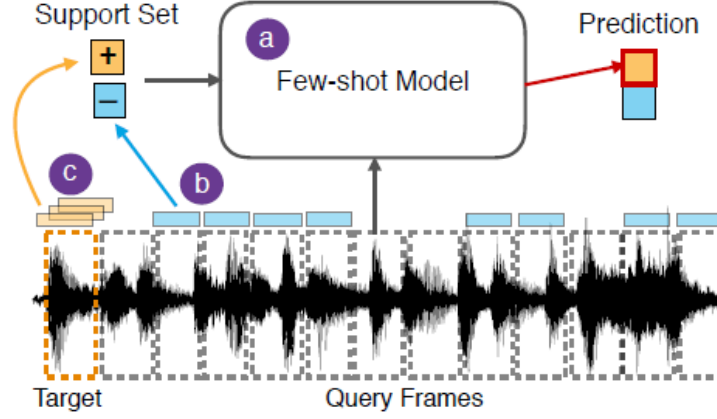


Figura 1: Metodo proposto per il few-shot sound event detection. (a) Applicazione del modello few-shot, (b) costruzione del set di esempi negativi, in blu, e (c) *data augmentation* per la generazione di più esempi positivi, in arancione. [6]

di embedding utilizzando una rete neurale  $f_{emb}$  e classificare correttamente ogni query con la funzione  $g_{sim}$ , che misura la somiglianza tra gli embedding di support e query.

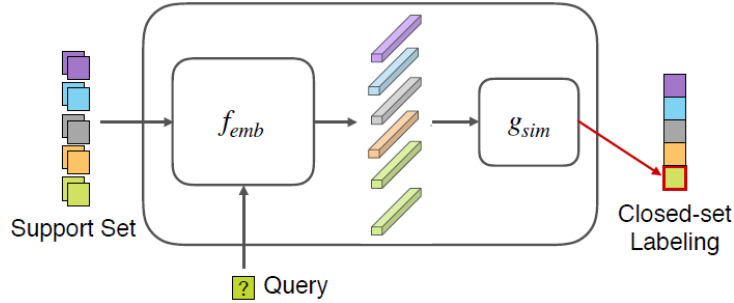


Figura 2: Modello few-shot learning nel caso 5-way 2-shot. [6]

Nell'articolo [6], vengono proposti quattro metodi di apprendimento basati sulla metrica per il rilevamento di eventi sonori e ne vengono confrontate le prestazioni. I modelli proposti sono: Siamese Networks [1], Matching Networks [5], Prototypical Networks [3] e Relation Networks [4].

La principale differenza tra questi metodi è la funzione  $g_{sim}$ , cioè la metrica della distanza utilizzata per misurare la somiglianza tra gli embedding del support e query set. Le Siamese Networks vengono addestrate utilizzando coppie di dati della stessa classe o di classi diverse con una triplet loss e la distanza L1. Le Matching Networks calcolano il coseno di similitudine tra

l'embedding della query e ogni support. Le Prototypical Networks calcolano il quadrato della distanza euclidea tra l'embedding della query e la media di ogni classe (prototipo) degli embedding del support set. Le Relation Networks sostituiscono le metriche a distanza fissa con un modulo di relazione composto a sua volta da una rete neurale.

In questa tesina sono state implementate la Prototypical Network e la Relation Network, comparandone i risultati nel capitolo 9. Nel capitolo 2 vengono introdotte le reti neurali convoluzionali usate per implementare la funzione  $f_{emb}$ , in 3 vengono descritte le reti few-shot più in particolare, mentre in 4 e 5 le due reti utilizzate. Nel capitolo 6 viene spiegato come è stato fatto il preprocessing del dataset, in 7 e 8 vengono descritte le implementazioni delle due reti usando *PyTorch*.

## 2 Convolutional Neural Network (CNN)

Una rete neurale convoluzionale (CNN) è una rete feedforward pensata per applicazioni che richiedono l'elaborazione di immagini o di grandi moli di dati. In una CNN, l'input è un tensore di dimensione: (numero di immagini)  $\times$  (altezza dell'immagine)  $\times$  (larghezza dell'immagine)  $\times$  (canali dell'immagine). Dopo aver attraversato un layer convoluzionale, l'immagine viene astratta in una *feature map* di dimensione: (numero di input)  $\times$  (altezza della feature map)  $\times$  (larghezza della feature map)  $\times$  (canali della feature map).

Un layer convoluzionale all'interno di una CNN è costituito da:

- filtri/kernel convoluzionali definiti da una larghezza e un'altezza.
- il numero di canali di ingresso e di uscita. I canali di input di un layer devono essere uguali al numero di canali di output dell'input.
- ulteriori funzioni dell'operazione di convoluzione, come: *padding*, *stride* e *dilatation*.

Il parametro stride definisce il passo con cui il kernel viene traslato sulla matrice di ingresso. Solitamente, viene effettuato uno zero-padding sui contorni del volume in modo da poter caratterizzare anche i valori che si trovano ai bordi delle matrici. In seguito alla convoluzione viene applicata una funzione non-lineare, come ad esempio una sigmoide o una ReLu, con lo scopo di aumentare la proprietà di non linearità.

Il pooling layer è uno strato della rete che ha lo scopo di ridurre le dimensioni della matrice prodotta dal convolutional layer. Combinando gruppi di elementi della matrice (solitamente  $2 \times 2$ ) restituisce il valore massimo tra essi, che andrà a sostituire il blocco stesso. La figura 3 illustra un esempio di max pooling  $2 \times 2$ .

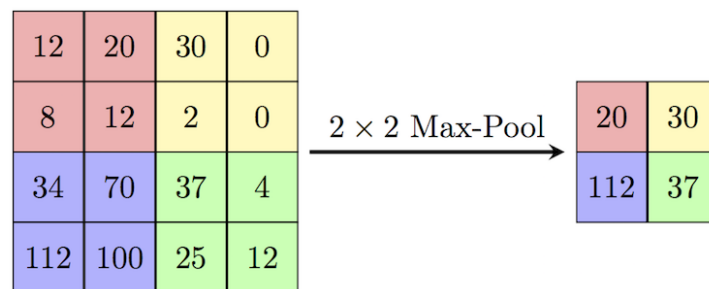


Figura 3: Esempio di max pooling  $2 \times 2$ .

## 3 Few-shot learning

### 3.1 Introduzione

Gli esseri umani possono riconoscere nuove classi di oggetti partendo da pochissimi esempi. Tuttavia, la maggior parte delle tecniche di machine learning richiedono migliaia di esempi per ottenere prestazioni simili a quelle umane. L'obiettivo del *few-shot learning* è classificare i nuovi dati dopo aver visto solo pochi esempi di training. Nel caso estremo, potrebbe esserci solo un singolo esempio per ogni classe (*one shot learning*). Nelle applicazioni pratiche, il few-shot learning è utile quando è difficile trovare esempi di training (ad es. casi di una malattia rara) o quando l'etichettatura dei dati è onerosa.

L'apprendimento few-shot viene solitamente studiato utilizzando la classificazione *C-way K-shot*. L'obiettivo è quello di discriminare le  $C$  classi composte da  $K$  esempi ciascuna. Una tipica dimensione del problema potrebbe essere quella di discriminare tra  $C = 10$  classi con solo  $K = 5$  esempi ciascuna durante il training. Questo tipo di apprendimento viene usato quando non possiamo allenare un classificatore usando metodi convenzionali, poiché qualsiasi algoritmo di classificazione moderno dipenderà da molti più parametri rispetto agli esempi di addestramento e generalizzerà male.

Se i dati non sono sufficienti per ridurre il problema, una possibile soluzione è acquisire esperienza da altri problemi simili. A tal fine, la maggior parte degli approcci caratterizza l'apprendimento a breve termine con un problema di meta-apprendimento.

### 3.2 Struttura del meta learning

Nell'apprendimento classico, impariamo come classificare dai dati di training e valutiamo i risultati utilizzando i dati di test. Nel quadro del meta-apprendimento, *impariamo come imparare* a classificare durante una serie di *episodi* di training e valutiamo le performance durante una serie di episodi di test. In altre parole, usiamo un insieme di problemi di classificazione per aiutare a risolvere altri insiemi non correlati.

Per la classificazione *C-way K-shot*, ogni task include  $C$  classi con  $K$  esempi ciascuna. Questi sono chiamati *support set* e vengono utilizzati per apprendere come risolvere il task. Inoltre, esistono ulteriori esempi delle stesse classi, note come *query set*, che non fanno parte del support set, utilizzate per valutare le prestazioni dell'episodio corrente. Ogni episodio può essere completamente unico: potremmo non vedere mai le classi di un episodio in nessuno degli altri. L'idea è che il sistema veda ripetutamente istan-

ze durante l'addestramento che corrispondono alla struttura del task finale dell'algoritmo di few-shot, ma che contengono classi diverse.

Ad ogni fase del meta-apprendimento, aggiorniamo i parametri in ogni episodio di training usando i dati di allenamento selezionati casualmente. La funzione di loss è determinata dalle prestazioni di classificazione sul set di query dell'episodio, in base alla conoscenza acquisita dal relativo support set. Poiché la rete viene sottoposta ad un compito diverso in ogni fase temporale, deve imparare a discriminare le classi di dati in generale, piuttosto che un particolare sottoinsieme di classi.

Per valutare le prestazioni nel few-shot, utilizziamo una serie di dati di test. Il set di test contiene solo classi che non erano nel training set. Per ciascuno, misuriamo le prestazioni sul query set in base alla conoscenza del *positive set* e del *negative set*. Il positive set è composto dalle stesse istanze della parola del query set, mentre il negative set è composto da parti di audio prese casualmente nella registrazione che non fanno parte del positive set.



## 4 Prototypical Network

Le Prototypical Network apprendono una rappresentazione spaziale in cui la classificazione può essere eseguita calcolando le distanze delle rappresentazioni del *prototipo* di ciascuna classe.

Per fare ciò, apprendiamo un mapping non lineare dell’input in uno spazio di embedding utilizzando una rete neurale e prendiamo il prototipo di una classe come la media del suo support set nello spazio di embedding. La classificazione viene quindi eseguita per un punto dell’embedding del query semplicemente trovando il prototipo della classe più vicino. La figura 4 mostra come avviene la classificazione della query  $\mathbf{x}$  avendo costruito tre prototipi  $\mathbf{c}_1$ ,  $\mathbf{c}_2$  e  $\mathbf{c}_3$ .

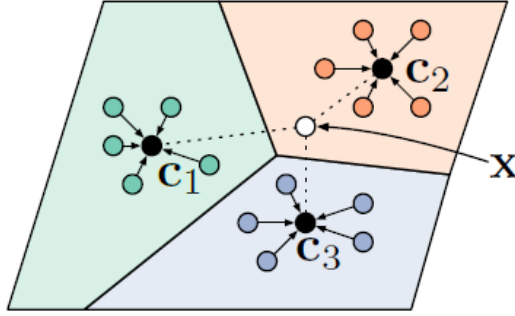


Figura 4: I prototipi few-shot  $\mathbf{c}_k$  sono calcolati come la media degli embedding del support set per ogni classe. I punti degli embedding delle query sono classificati facendo il softmax sulle distanze del prototipo delle classi:  $p_\phi(y = k|\mathbf{x}) \propto \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))$ . [3]

Nella classificazione few-shot per ogni episodio viene fornito un support set di  $N$  esempi etichettati  $S = \{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$  dove  $\mathbf{x}_i \in \mathbb{R}^D$  rappresenta il vettore  $D$ -dimensionale della feature (nel nostro caso spettrogrammi di dimensione  $128 \times 51$ ) e  $y_i \in \{1, \dots, K\}$  la rispettiva label.  $S_k$  denota il set di esempi etichettati con la classe  $k$ .

La rete Prototypical calcola una rappresentazione  $M$ -dimensionale  $\mathbf{c}_k$ , o *prototipo*, di ogni classe tramite una funzione di embedding  $f_\phi : \mathbb{R}^D \rightarrow \mathbb{R}^M$  con parametri da allenare  $\phi$ . La funzione di embedding è rappresentata da una rete convoluzionale. Ogni prototipo è calcolato come la media tra gli embedding delle istanze della stessa classe.

$$\mathbf{c}_k = \frac{1}{|S_k|} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_\phi(\mathbf{x}_i) \quad (1)$$

Data una funzione distanza  $d : \mathbb{R}^M \times \mathbb{R}^M \rightarrow [0, +\infty)$ , la rete Prototypical calcola la relazione di una query  $\mathbf{x}$  rispetto ai prototipi tramite la funzione softmax delle distanze prese con segno negativo.

$$p_\phi(y = k|\mathbf{x}) = \frac{\exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))}{\sum_k \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}'_k))} \quad (2)$$

Il processo di training avviene minimizzando il negativo del logaritmo della probabilità

$$J(\phi) = -\log(p_\phi(y = k|\mathbf{x})) \quad (3)$$

considerando la distanza fra query e il prototipo della sua classe. Gli episodi di training sono formati campionando  $C$  classi di parole da un lettore e selezionando per ognuna casualmente  $K$  istanze e  $Q$  query.

Le reti Prototypical differiscono dalle reti Marching nel caso few-shot, mentre sono uguali nel caso one-shot. Le reti Marching producono un classificatore nearest neighbor ponderato dato il support set, mentre le reti Prototypical producono un classificatore lineare quando viene utilizzata la distanza euclidea al quadrato. Nel caso dell'apprendimento one-shot,  $\mathbf{c}_k = \mathbf{x}_k$  poiché esiste un solo punto di supporto per classe, in questo modo le reti Marching e le reti Prototypical diventano equivalenti.

Come nella maggior parte dei modelli di apprendimento few-shot, la funzione di embedding è composta da quattro blocchi convoluzionali. Ogni blocco è formato da 64 filtri  $3 \times 3$ , un layer di batch normalization, un layer ReLu che rappresenta la non linearità e un max-pooling  $2 \times 2$ .

## 5 Relation Network

La rete Relation ha lo scopo di associare due istanze alla volta per determinare la loro similarità. Questo viene effettuato concatenando gli embedding di più istanze in un unico elemento che sarà dato in ingresso a una rete decisionale i cui parametri saranno aggiornati in modo che una concatenazione di elementi simili restituisca un risultato vicino a 1.

La Relation Network è costituita da due moduli: un modulo di *embedding*  $f_\varphi$  (equivalente a quello nella Prototypical) e un modulo di *relation*  $g_\phi$ , come illustrato in figura 5. Le istanze  $x_i$  del query set  $\mathcal{Q}$  e quelle  $x_j$  del support set  $\mathcal{S}$  vengono date in ingresso al modulo di embedding producendo dei vettori (feature maps)  $f_\varphi(x_i)$  e  $f_\varphi(x_j)$ .

Questi ultimi vengono poi dati all'operatore  $\mathcal{C}(\cdot, \cdot)$  che ne fa la concatenazione:  $\mathcal{C}(f_\varphi(x_i), f_\varphi(x_j))$ . Le feature map concatenate passano poi attraverso il modulo di decisione che restituisce uno scalare da 0 a 1, il quale rappresenta la somiglianza tra  $x_i$  e  $x_j$ .

Per il caso  $C$ -way one-shot, viene concatenata la query con le istanze delle  $C$  classi producendo  $C$  punteggi di somiglianza chiamati  $r_{i,j}$ .

$$r_{i,j} = g_\phi(\mathcal{C}(f_\varphi(x_i), f_\varphi(x_j))), \quad i = 1, 2, \dots, C \quad (4)$$

Nel caso  $C$ -way K-shot invece, la query viene concatenata con la somma elemento per elemento degli embedding di ogni istanza delle classi. Quindi, in entrambi i casi i confronti  $r_{i,j}$  sono  $C$  per ogni query.

Per allenare il modello viene usato l'errore quadratico medio (MSE) in modo che l'uscita del modulo di decisione produca 1 se i vettori concatenati sono della stessa classe e 0 altrimenti.

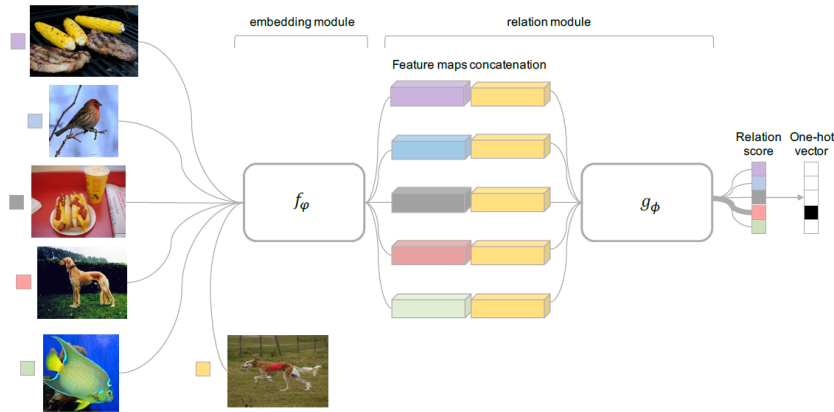


Figura 5: Architettura della Relation Network nel caso *5-way 1-shot* con un esempio di query. [4]

Come nella Prototypical Network, anche nella Relation Network viene usata una architettura con quattro blocchi convoluzionali per il modulo di embedding, come illustrato in figura 6.

Ogni blocco convoluzionale contiene una convoluzione con 64 filtri  $3 \times 3$ , una batch normalization e un layer ReLU non lineare. I primi tre blocchi contengono anche un layer di max pooling  $2 \times 2$ , mentre l'ultimo no. Lo facciamo perché abbiamo bisogno delle feature map di output per gli ulteriori livelli convoluzionali nel modulo realtion.

Il modulo realtion è costituito da due blocchi convoluzionali e due layer fully-connected. Ciascuno dei blocchi convoluzionali è una convoluzione  $3 \times 3$  con 64 filtri seguita da batch normalization, non linearità ReLU e da un max pooling  $2 \times 2$ . La dimensione di output dell'ultimo layer di max pooling è pari a 64. I due layer fully-connected sono rispettivamente 8 e 1 dimensionale. Tutti i layer fully-connected sono ReLU eccetto quello di output, composto da una sigmoide per generare punteggi di relazione in un intervallo compreso tra 0 e 1.

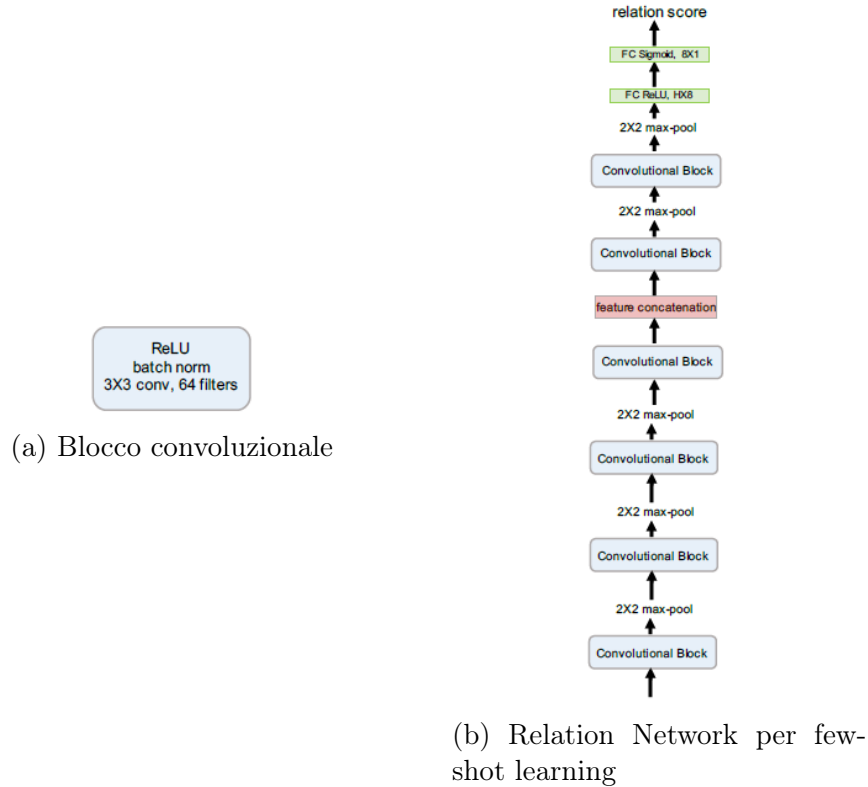


Figura 6: Architettura della Relation Network per few-shot learning (b) composta dagli elementi inclusi nel blocco convoluzionale (a). [4]

## 6 Preprocessing del dataset

### 6.1 Dataset Spoken Wikipedia Corpora

Il progetto Spoken Wikipedia unisce lettori volontari di articoli di Wikipedia. Sono disponibili centinaia di articoli in inglese, tedesco e olandese per gli utenti che non sono in grado o non vogliono leggere la versione scritta dell'articolo. Il dataset trasforma i file audio in un corpus, cioè una raccolta ordinata e completa di opere o di autori, allineato nel tempo, rendendolo accessibile per la ricerca.[2]

Questo corpus ha diverse caratteristiche importanti:

- centinaia di ore di audio allineato
- lettori eterogenei
- diversi argomenti
- genere testuale ben studiato
- le annotazioni delle parole possono essere mappate all'html originale
- allineamenti a livello di fonema

Ogni articolo è suddiviso in sezioni, frasi e token. Ogni token è normalizzato e la normalizzazione è allineata all'audio.

### 6.2 Le annotazioni del dataset

Ogni annotazione è racchiusa in un tag chiamato “**article**”, che contiene una sezione “**meta**” per i metadati, come si può vedere in 1 e una sezione “**d**”, come in 2, contenente l'articolo e le relative annotazioni.

```
<article>
  <meta>
    <link key="DC.conformsto" value="http://nats.gitlab.io/swc/schema/swc-1.0.rnc"/>
    <prop key="DC.creator" value="Spoken Wikipedia Corpus Collection Software"/>
    <prop key="DC.publisher" value="Universität Hamburg"/>
    <link key="DC.reference" value="http://nbn-resolving.de/urn:nbn:de:gbv:18-228-7-2209"/>
    <prop key="DC.type" value="dataset"/>
    <prop key="DC.license" value="CC-BY-SA"/>
    <prop key="DC.title" value="Limerence"/>
    <prop key="DC.language" value="en"/>
    <prop key="DC.identifier" value="Limerence"/>
    <prop key="DC.date.read" value="2005-04-29 00:00:00"/>
    <link key="DC.source" value="https://en.wikipedia.org/wiki/Limerence"/>
    <prop key="DC.source.wikiID" value="154147"/>
    <prop key="DC.source.revision" value="791627969"/>
    <link key="DC.source.text" value="https://en.wikipedia.org/w/index.php?title=Limerence&oldid=13811989"/>
    <link key="DC.source.audio" value="https://upload.wikimedia.org/wikipedia/commons/aa/Limerence.ogg" group="audiol"/>
    <prop key="DC.source.audio.offset" value="0.0" group="audiol"/>
    <link key="DC.source.audio.page" value="https://en.wikipedia.org/w/index.php?title=File%3aLimerence.ogg" group="audiol"/>
```

```

<link key="DC.source.audio.date" value="2007-09-14 19:23:05" group="audio1"/>
<prop key="reader.name" value="the Epopt"/>
<prop key="processing.step" value="tokenize" group="tokenize"/>
<prop key="processing.step.date" value="2017-08-10T07:44:44.095+02:00[Europe/Berlin]"
      group="tokenize"/>
<prop key="processing.step.options" value="Namespace(output=articles/Limerence/
tokenized.swc, all_sections=false, null_normalize=false, raw_output=null,
subparser_name=tokenize, lang=en, no_introduction=false, article_dir=articles/
Limerence)" group="tokenize"/>
<prop key="processing.step.git.commit.id" value="7431
dbf93f212ad828208abaf8f518fb8de11ff3" group="tokenize"/>
<prop key="processing.step.git.commit.time" value="09.08.2017 @ 15:21:55 CEST" group
="tokenize"/>
<prop key="processing.step" value="align" group="align"/>
<prop key="processing.step.date" value="2017-08-11T16:12:18.423+02:00[Europe/Berlin]"
      group="align"/>
<prop key="processing.step.options" value="Namespace(output=articles/Limerence/
aligned.swc, transcript=articles/Limerence/tokenized.swc, g2p=../model_en/model.
fst.ser, phone=false, subparser_name=align, dict=../model_en/empty.dic,
acoustic_model=../model_en/, audio=articles/Limerence/audio.wav)" group="align"
/>
<prop key="processing.step.git.commit.id" value="7431
dbf93f212ad828208abaf8f518fb8de11ff3" group="align"/>
<prop key="processing.step.git.commit.time" value="09.08.2017 @ 15:21:55 CEST" group
="align"/>
</meta>

```

Codice 1: Metadati delle annotazioni delle parole in un audio

Il documento, contrassegnato dal tag “d”, può contenere parti diverse, ciascuna spiegata di seguito. La sezione “extra” contiene il testo che abbiamo incluso ma non fa parte dell’articolo, “ignored” contiene ciò che fa parte del testo ma viene ignorato per l’allineamento, “section” contiene un titolo e un contenuto, “p” contiene un paragrafo e “s” una frase che a sua volta contiene dei token “t”. In quest’ultimo è contenuta la singola parola originale e le normalizzazioni. Per esempio, la punteggiatura non ha annotazioni di normalizzazione in quanto non è pronunciata, ma il numero 500 ne ha due - “cinque” e “cento”. un token stesso non ha allineamento, solo la sua normalizzazione “n” è allineata. La normalizzazione ha una “pronunciation” e può avere un tempo di “start” ed “end”, se è allineata. La normalizzazione, a sua volta, può contenere dei fonemi “ph”.

```

<d>
<extra text="LimerenceFrom wikipedia, the free encyclopedia at e n dot wikipedia dot
org.">
<s text="LimerenceFrom wikipedia, the free encyclopedia at e n dot wikipedia dot org
.">
<t text="LimerenceFrom">
<n pronunciation="LimerenceFrom" start="140" end="1190"/>
</t>
<t text="wikipedia">
<n pronunciation="wikipedia" start="1250" end="1950"/>
</t>
<t text=","/>
<t text="the">
<n pronunciation="the" start="1950" end="2070"/>
</t>
<t text="free">
<n pronunciation="free" start="2070" end="2300"/>
</t>
<t text="encyclopedia">
<n pronunciation="encyclopedia" start="2300" end="3220"/>
</t>
<t text="at">
<n pronunciation="at"/>
</t>
<t text="e">
<n pronunciation="e" start="3490" end="3710"/>
</t>

```

### 6.3 Lettori del dataset

Il dataset Spoken Wikipedia Corpora contiene un totale di 1340 audio di diversi lettori, ma nel progetto vengono presi solo gli audio che contengono annotazioni a livello di parola. In questo modo vengono presi solo 208 lettori e partizionati come in [6] con un rapporto 138 : 15 : 30 tra lettori di training, validation e test. I lettori e le parole sono state estratte dai file “aligned.swc” contenuti in ogni audio e, successivamente, salvati in diversi file json. Per la gestione dei json sono state create due funzioni utili per la lettura e scrittura come mostrato nel codice 3.

```
1 import json
2
3 def write_json_file(filename, list_of_dict, indent = 0):
4     f = open(filename, "w")
5     f.write(json.dumps(list_of_dict, indent = indent))
6     f.close
7
8 def read_json_file(filename):
9     f = open(filename, "r")
10    list_of_dict = json.load(f)
11    f.close
12    return list_of_dict
```

Codice 3: json\_manager.py

Il codice 4 salva un file json chiamato “readers\_paths.json” formato dalle chiavi “reader\_name” e “folder” i cui valori sono rispettivamente il nome del lettore e le cartelle in cui sono salvati i file audio registrati dal relativo lettore, come si può vedere nel json 5. Questo codice, usando la libreria `xml.etree.ElementTree` e la funzione `ET.parse`, rappresenta l’intero documento XML come un albero. La funzione `getroot` ne trova la radice e, successivamente, si scorre l’albero iterando finché non si trova il tag “prop”, in cui è contenuta la chiave “reader.name”. Inoltre, si effettua un controllo sul nome del lettore perché può capitare che questo venga salvato nel file “aligned.swc” in modi diversi. Ad esempio, in alcuni file si può trovare nella chiave “reader.name” il valore “:en:user:alexkillby|alexkillby”, mentre in altri solo “alexkillby” oppure si può trovare “user:popularoutcast”, mentre in altri solo “popularoutcast”. Una volta noto il nome del lettore si vede se è già presente nella lista di dizionari e se è un lettore nuovo lo si aggiunge con il relativo nome del file audio, mentre se il lettore era già presente si aggiunge solamente il nome del file audio.

```

1 import xml.etree.ElementTree as ET
2 from tqdm import tqdm
3 import os
4 import json
5 from json_manager import *
6
7 # Initialize list with empty dictionaries
8 readers = []
9
10 source_path = "./Dataset/English spoken wikipedia/english/"
11 filename = "aligned.swc"
12
13 # search for readers for each folder
14 for audio_path in os.scandir(source_path):
15     # save only folder name from entire path
16     folder = os.path.basename(audio_path)
17     if (os.path.exists(source_path + "/" + folder + "/" + filename)):
18         # parse the xml file "aligned.swc"
19         tree = ET.parse(source_path + "/" + folder + "/" + filename)
20         # getroot returns the root element for this tree
21         root = tree.getroot()
22         #root.iter creates a tree iterator with the current element as the root.
23         # The iterator iterates over this element and all elements below it, in
24         # document (depth first) order.
25         for property in root.iter(tag = 'prop'):
26             # if the key "reader.name" exists
27             if (property.attrib['key'] == 'reader.name'):
28                 # save the reader name taking the value of the attribute
29                 reader_name = property.attrib['value'].lower()
30                 # fix readers names that contain "user:"
31                 if ("user:" in reader_name):
32                     # fix readers names that contain "|"
33                     if ("|" in reader_name):
34                         # example reader_name = [[en:user:alexkillby|alexkillby]] ->
35                         # reader_name = alexkillby
36                         reader_name = reader_name[reader_name.find("user:") + 5:reader_name.find(
37                             "|")]
38                     # fix readers names that contain "|"
39                 elif ("]]" in reader_name):
40                     # example reader_name = [[user:popularoutcast]] ->
41                     # reader_name = popularoutcast
42                     reader_name = reader_name[reader_name.find("user:") + 5:reader_name.find(
43                         "]]")]
44             # if the reader is not yet on the list create a dict and append to
45             # the readers list
46             if not any(reader['reader_name'] == reader_name for reader in readers):
47                 dictionary = {'reader_name': reader_name, 'folder': [folder]}
48                 readers.append(dictionary)
49             else:
50                 # if the reader is already on the list add the folder name
51                 for reader in readers:
52                     if (reader['reader_name'] == reader_name):
53                         reader['folder'].append(folder)
54 # print the number of the readers
55 print("The readers are:", str(len(readers)))
56
57 # save a "readers_paths.json" with the name of the readers and the relative
58 # file audio folders
59 write_json_file("readers_paths.json", readers, indent = 4)

```

Codice 4: xml\_parser\_readers.py



```

{
  "reader_name": "the epopt",
  "folder": [
    "(I_Can%27t_Get_No)_Satisfaction",
    "Ceremonial_ship_launching",
    "Limerence",
    "Revolt_of_the_Admirals",
    "Ship_commissioning"
  ]
},
{
  "reader_name": "wodup",
  "folder": [
    "0.999..%2e",
    "Execution_by_elephant",
    "Hell_Is_Other_Robots",
    "Tom_Bosley",
    "Truthiness"
  ]
},

```

Codice 5: Formato del file readers\_paths.json

## 6.4 Parole degli audio

Il codice 6 mostra come viene creato il json 7, formato dalle chiavi “word”, “frequency”, “start” ed “end”, le quali, a loro volta, contengono la parola, il numero di volte in cui è stata ripetuta e il timestamp di inizio e fine.

Per ogni cartella del dataset SWC, se esistono i file “audio.ogg” e “aligned.swc”, si procede con il parsing del file XML, iterando l’albero fino al tag “n”, cioè fino al tag che contiene la normalizzazione della parola come spiegato in 6.2. Se è presente la chiave “start” (o, in modo equivalente, “end”), la parola viene aggiunta alla lista **words**.

Per trovare quante volte viene ripetuta la singola parola abbiamo usato la sottoclasse **Counter** della classe **dict** di Python, la quale restituisce una raccolta in cui gli elementi sono le chiavi del dizionario e il numero di ripetizioni è il loro valore. Si costruisce dunque una lista di dizionari chiamata **target\_words**, che contiene le *target words*, cioè le parole che si ripetono almeno 10 volte nel testo. [6]

Si rifà un parsing del file XML per aggiungere i timestamp di “start” e “end” ad ogni parola. Il file “word\_count.json” contiene tutte le parole che si ripetono almeno 10 volte e viene salvato all’interno di ogni cartella del dataset. Infine, si filtrano le target words, utili poi in fase di test della rete: se ci sono più di 10 parole che soddisfano la condizione di target word, si prendono tra queste solo 10 scelte in modo random. In questo modo, si cerca di evitare di scegliere parole molto comuni o molto rare. Le target

words vengono salvate nel file “target\_words.json” contenuto all’interno di ogni cartella del dataset.

```

1  import xml.etree.ElementTree as ET
2  from tqdm import tqdm
3  import os
4  import collections
5  from json_manager import *
6  import random
7
8  folder = []
9  source_path = "./Dataset/English spoken wikipedia/english/"
10 filename = "aligned.swc"
11 file_audio_name = "audio.ogg"
12
13 # iterate in each folder of the dataset
14 for folder in tqdm(os.scandir(source_path), desc = "Folder number"):
15     # initialize the list of dict "target_words"
16     target_words = []
17     if (os.path.exists(folder.path + "/" + filename) \
18         and os.path.exists(folder.path + "/" + file_audio_name)):
19         # parse the xml file aligned.swc
20         tree = ET.parse(folder.path + "/" + filename)
21         # getroot returns the root element for this tree
22         root = tree.getroot()
23         # initialize the list "words"
24         words = []
25         # root.iter creates a tree iterator with the current element as the
26         # root. The iterator iterates over this element and # all elements
27         # below it, in document (depth first) order.
28         for token_normalization in root.iter(tag = 'n'):
29             # we take only the words with the timestamp, so only if there is
30             # 'start' (or 'end') tag
31             if 'start' in token_normalization.keys():
32                 # add every word with "start" key
33                 words.append(token_normalization.attrib['pronunciation'].
34                             lower())
35
36         # collections.Counter stores elements as dictionary keys, and their
37         # counts are stored as dictionary values.
38         unique_words = collections.Counter(words)
39         # for each key (word) in "unique_words" append a new target_word if
40         # the number of occurency is at least 10
41         for key in unique_words.keys():
42             # we only consider words that occur at least 10 times in the
43             # recording. Note that unique_words[key] is the word frequency
44             if (unique_words[key] >= 10):
45                 # add a new target word
46                 target_words.append({'word' : key, \
47                                     'frequency' : unique_words[key], \
48                                     'start' : [], \
49                                     'end' : []})
50
51         # for each "target_words" append the relative "start" and "end"
52         # timestamp
53         for token_normalization in root.iter(tag = 'n'):
54             # we take only the words with the timestamp, so only if there is
55             # 'start' (or 'end') tag
56             if 'start' in token_normalization.keys():
57                 # iterate over the "target_words"
58                 for target_word in target_words:
59                     # add start and end timestamp only to the relative
60                     # "target_word"

```

```

57         if target_word['word'] == token_normalization.attrib['
           pronunciation'].lower():
58             target_word['start'].append(int(token_normalization.
           attrib['start']))
59             target_word['end'].append(int(token_normalization.
           attrib['end']))
60
61     write_json_file(folder.path+"/word_count.json", target_words, indent
           = 4)
62     # If there are more than 10 words that occur at least 10 times in
63     # the recording, we sort the words by their number of occurrences,
64     # divide the sorted list into 10 equally sized bins, and sample
65     # one keyword per bin.
66     if (len(target_words) >= 10):
67         target_words = random.sample(target_words, 10)
68     # save the "target_words.json"
69     write_json_file(folder.path+"/target_words.json", target_words,
           indent = 4)

```

Codice 6: find\_target\_words.py

```

{
    "word": "i",
    "frequency": 12,
    "start": [
        660,
        8800,
        115050,
        ...
    ],
    "end": [
        870,
        8940,
        115240,
        ...
    ]
},
{
    "word": "the",
    "frequency": 75,
    "start": [
        4160,
        49930,
        53680,
        ...
    ],
    "end": [
        4320,
        50030,
        53710,
        ...
    ]
},
...
}

```

Codice 7: Formato del file word\_count.json

## 6.5 Associazione delle parole con il rispettivo lettore

I file “readers\_paths.json” e “word\_count.json” vengono successivamente usati per salvare le parole di ogni lettore, il nome della relativa cartella in cui vengono pronunciate e i timestamp di inizio e fine della parola per ogni cartella come mostrato nel json 9. Il codice 8 legge il nome di ogni lettore dal json precedentemente salvato, successivamente per ognuna delle cartelle del lettore viene usato “word\_count.json” per ottenere tutte le parole dell’audio e i relativi timestamp.

Viene salvato, infine, il file “readers\_words.json” che contiene per ogni lettore le parole pronunciate, per ognuna di queste l’audio in cui vengono enunciate e per ogni audio i timestamp di inizio e fine.

```
1  from tqdm import tqdm
2  import os
3  from json_manager import *
4
5  source_path = "./Dataset/English spoken wikipedia/english/"
6  # read the json that contains the readers name and their audio folders
7  readers = read_json_file("readers_paths.json")
8  # initialize a list of dict
9  readers_words = []
10 # for each reader search the word spoken by the reader
11 for reader in tqdm(readers):
12     # create a dict with 'reader_name' and 'words' keys
13     new_readers_words = {'reader_name' : reader['reader_name'], \
14                          'words' : [] }
15
16     # for each reader create a new dict for words
17     words_per_reader = []
18     # flag to signal if "word_count.json" exists
19     json_file_exist = False
20     # search for each folder the file "word_count.json"
21     for reader_folder in reader['folder']:
22         if (os.path.exists(source_path + "/" + reader_folder + "/word_count.
23                               json")):
24             # "word_count.json" exists
25             json_file_exist = True
26             # read "word_count.json"
27             recording_words = read_json_file(source_path + "/" +
28                                               reader_folder + "/word_count.json")
29             # for each audio folder create a new list of dict
30             folder_per_word = []
31             # for each word in the audio save the folder, start and end
32             # timestamps
33             for word in recording_words:
34                 # create a dict with 'folder', 'start' and 'end' keys
35                 folder_per_word = {'folder' : reader_folder, \
36                                    'start' : word['start'], \
37                                    'end' : word['end']}
38                 # if the word is not yet in the list add the word
39                 if not any (word['word'] == word_per_reader['word'] for
40                             word_per_reader in words_per_reader):
41                     words_per_reader.append({'word' : word['word'], \
42                                              'folders' : [folder_per_word]})
43
44     # otherwise add 'start' and 'end' if the "reader_folder" is
```

```

41         # in word_per_reader['folders'] or add the "folder_per_word"
42         else:
43             for word_per_reader in words_per_reader:
44                 if word['word'] == word_per_reader['word']:
45                     if reader_folder in word_per_reader['folders']:
46                         for folder in word_per_reader['folders']:
47                             if reader_folder == folder['folder']:
48                                 folder['start'] += word['start']
49                                 folder['end'] += word['end']
50                     else:
51                         word_per_reader['folders'].append(
52                             folder_per_word)
53     # if "word_count.json" exists add the new reader to "training_readers"
54     # list
55     if (json_file_exist):
56         new_readers_words['words'] = words_per_reader
57         readers_words.append(new_readers_words)
58 # save a "readers_words.json" with the name of the readers and the relative
59 # words spoken
60 write_json_file("readers_words.json", readers_words, indent = 0)

```

Codice 8: words\_per\_reader.py

```

"reader_name": "wodup",
"words": [
    {
        "word": "the",
        "folders": [
            {
                "folder": "0.999..%2e",
                "start": [
                    6950,
                    1029740,
                    1032520,
                    ...
                ],
                "end": [
                    7190,
                    1029880,
                    1032620,
                    ...
                ]
            },
            {
                "folder": "Execution_by_elephant",
                "start": [
                    3600,
                    10680,
                    ...
                ],
                "end": [
                    ...
                ]
            }
        ]
    },
    ...
]

```

Codice 9: Formato del file readers\_words.json

## 6.6 Calcolo dello spettrogramma delle parole

Per ogni istanza delle parole, prendiamo una finestra di mezzo secondo centrata sulla parola, calcoliamo lo spettrogramma mel da 128 bin e lo portiamo scala logaritmica. Lo spettrogramma rappresenta l'intensità di un suono in

funzione del tempo e della frequenza. Sull'asse delle ascisse della rappresentazione grafica dello spettrogramma è riportato il tempo, sull'asse delle ordinate la frequenza e per ogni punto del grafico un colore rappresenta l'intensità del suono.

La scala *mel* è una scala di percezione dell'altezza (pitch) di un suono. La relazione tra la scala mel e quella comunemente usata è rappresentata dall'uguaglianza tra 1000 Mel e 1000 Hz all'intensità di 40 dB. La scala mel è definita nel seguente modo

$$\text{mel} = 2595 \cdot \log \left( 1 + \frac{f}{700} \right) \quad (5)$$

e nella figura 7 si può vedere la sua funzione in relazione agli Hertz.

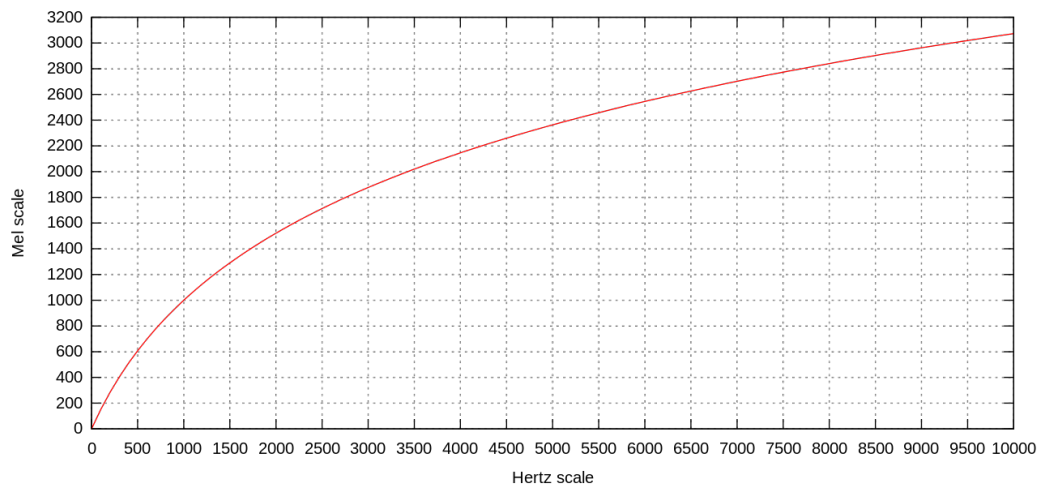


Figura 7: Grafico della scala mel in funzione della scala in Hertz.

Per il calcolo dello spettrogramma mel occorre dividere l'input in blocchi usando finestre con overlap, fare la FFT di ogni blocco, generare la scala mel e associare ogni componente spettrale alla relativa frequenza nella scala mel.

La funzione `compute_melspectrogram`, mostrata nel codice 10, usando la libreria *librosa*, calcola gli spettrogrammi da dare in input ai modelli della rete neurale. I file audio, originalmente campionati ad una frequenza di 44.1 kHz, sono sottocampionati a 16 kHz. Per calcolare lo spettrogramma, usiamo una lunghezza della finestra di 25 ms, un hop size di 10 ms e una FFT di 64 ms. Lo spettrogramma è contenuto in una matrice di dimensioni  $128 \times 51$ , le cui righe rappresentano le 128 bande mel, mentre le colonne contengono le FFT di ogni finestra da 10 ms.

```
1 import librosa
```

```

2
3 def compute_melspectrogram(filename, word_center):
4     # audio recordings are downsampled to a sampling rate of 16 kHz.
5     sample_rate = 16000
6     #librosa.load loads the audio file as a floating point time series.
7     # y: audio time series, sr: sample rate
8     y, _ = librosa.load(path = filename, sr = sample_rate, offset =
        word_center - 0.25, duration = 0.5)
9
10    # We use a window length of 25 ms, hop size of 10 ms and a fast Fourier
11    # transform size of 64 ms.
12    window_length = int(0.025*sample_rate)
13    hop_size = int(0.01*sample_rate)
14    fft_size = int(0.064*sample_rate)
15    # For each word instance, we take a half-second context window centered
16    # on the word and compute a 128 bin log-mel-spectrogram
17    mels_number = 128
18    # librosa.feature.melspectrogram computes a mel-scaled spectrogram.
19    mel_spectrogram = librosa.feature.melspectrogram(y, sr = sample_rate,
        n_fft = fft_size, hop_length = hop_size, win_length = window_length,
        n_mels = mels_number)
20    # librosa.power_to_db converts a power spectrogram to a dB-scale
21    # spectrogram.
22    log_mel_spectrogram = librosa.power_to_db(mel_spectrogram)
23
24    return log_mel_spectrogram

```

Codice 10: mel\_spectrogram.py

Il risultato della funzione `log_mel_spectrogram` applicato alla parola “stones” pronunciata tra 50.35s e 50.78s nell’audio contenuto nella cartella “I can’t get no satisfaction” è mostrato nella figura 8. Come ingresso della funzione viene dato il percorso del file “audio.ogg” contenuto nella cartella “I can’t get no satisfaction” e come centro della parola “stones” il tempo  $\frac{50.78+50.35}{2}$  s.

## 6.7 Creazione delle feature

Gli spettrogrammi descritti in 6.6 sono gli ingressi della rete neurale e sono stati salvati nel preprocessing prima di allenare la rete. Poiché per ogni episodio di training si usa un lettore campionato in modo random dal training set occorre dividere i lettori in training, validation e test. Infine, per ogni lettore occorre salvare gli spettrogrammi delle parole pronunciate. Il codice 11 mostra come dividere i lettori e salvare le varie feature. Tra tutti i lettori salvati nel file “readers\_words.json” vengono considerati validi solo quelli con almeno due classi, cioè due parole pronunciate e almeno 26 istanze per ogni parola. Questi numeri sono scelti perché in [6] vengono proposti diversi modelli a seconda del numero delle classi  $C$  e delle istanze  $K$  per ogni classe. Il minimo numero di  $C$  è pari a 2, mentre il numero minimo di istanze è pari a 26 perché nel training occorre utilizzare 16

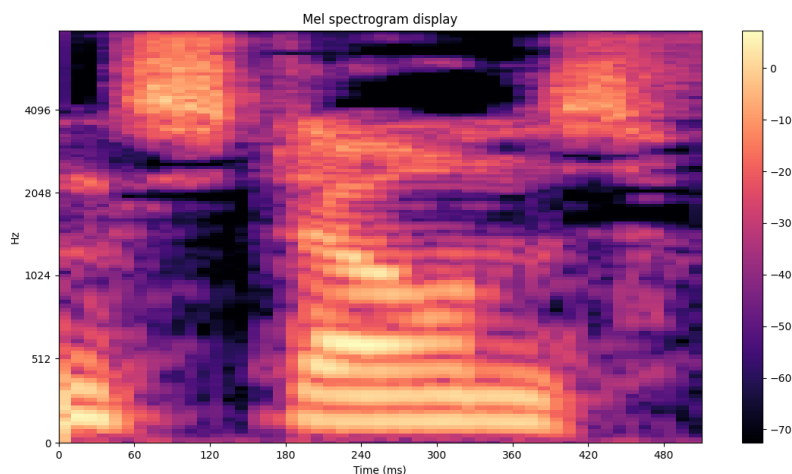


Figura 8: Spettrogramma di mezzo secondo centrato sulla parola “stones” pronunciata nell’audio “I can’t get no satisfaction” dopo 50.35 secondi.

istanze per la query e un numero variabile di  $K$  che nel caso peggiore è uguale a 10, quindi in totale servirebbero 16+10 istanze della parola. Una volta noti i lettori validi con la funzione `find_valid_readers`, questi vengono ordinati in maniera random prima di dividerli in due gruppi: lettori di training insieme a quelli di validation e lettori di test. La funzione che divide i lettori con un rapporto 138 : 15 : 30 tra training, validation e test si chiama `create_training_validation_test_readers`. I lettori di training e validation saranno poi divisi prima dell’inizio del training.

Le feature, cioè gli spettrogrammi, vengono salvati in due cartelle chiamate “Training\_validation\_features” e “Test\_features”, le quali hanno al loro interno delle cartelle rispettivamente chiamate come il nome del lettore e come il nome dell’articolo di Wikipedia. A sua volta, ogni cartella contiene gli spettrogrammi di ogni istanza della parola.

Il dataset di training e validation viene salvato con la funzione `save_dataset`, mentre quello di test con `save_test_dataset`. Il numero massimo di classi e di istanze per ogni classe salvate sono rispettivamente 32 e 64. Questi numeri sono stati scelti perché altrimenti il dataset sarebbe composto da un elevato numero di classi e istanze, risultando di dimensioni molto grandi.

```

262 min_classes = 2 # minimum number of classes
263 min_instances_per_class = 26 # minimum number of instances per class
264
265 valid_readers = find_valid_readers(min_classes, min_instances_per_class)
266 # random sample the valid readers
267 valid_readers = random.sample(valid_readers, len(valid_readers))
268

```



```

269 # The readers are partitioned into training, validation, and test sets with
    a
270 # 138:15:30 ratio
271 number_of_training_readers = int(138/183*len(valid_readers))
272 number_of_test_readers = int(30/183*len(valid_readers))
273 number_of_validation_readers = int(15/183*len(valid_readers))
274
275 # The valid readers are partitioned into training, validation, and test
276 # readers
277 training_validation_readers, test_readers =
    create_training_validation_test_readers(valid_readers,
        number_of_training_readers, number_of_test_readers,
        number_of_validation_readers)
278 # create the folder for the training and validation features
279 training_validation_feature_folder_name = "Training_validation_features/"
280
281 if not (os.path.exists(training_validation_feature_folder_name)):
282     try:
283         os.mkdir(training_validation_feature_folder_name)
284     except OSError as error:
285         print(error)
286
287 max_class_number = 32
288 max_instances_number = 64
289
290 dataset_path = "./Dataset/English spoken wikipedia/english/"
291 audio_file_name = "audio.ogg"
292
293 training_validation_readers = read_json_file("training_validation_readers.
    json")
294
295 save_dataset(training_validation_readers,
    training_validation_feature_folder_name, dataset_path, audio_file_name,
    max_class_number, max_instances_number)
296
297 reader_paths = read_json_file('readers_paths.json')
298
299 # create the folder for the test features
300 test_feature_folder_name = "Test_features/"
301
302 if not (os.path.exists(test_feature_folder_name)):
303     try:
304         os.mkdir(test_feature_folder_name)
305     except OSError as error:
306         print(error)
307
308 save_test_dataset(test_readers, reader_paths, test_feature_folder_name,
    dataset_path, audio_file_name)

```

Codice 11: preprocessing.py

Nella funzione `find_valid_readers` viene letto il file “`readers_words.json`” e viene creata una lista di dizionari con chiavi “`reader_name`”, “`word`”, “`start`”, “`end`” e “`folders`”. Ogni lettore contiene infatti le parole pronunciate, i relativi timestamp e la cartella in cui è contenuto l’audio che contiene la parola pronunciata. Il dizionario è creato in modo che l’*i*-esimo valore di “`start`” o “`end`” corrisponda all’*i*-esimo valore di “`folders`”. I lettori validi sono quelli che pronunciano almeno 2 parole per 26 volte.

```

10 def find_valid_readers(min_classes = 2, min_instances_per_class = 26):
11     """
12     find_valid_readers returns a list of dict with 'word', 'start', 'end'
13     and 'folders' keys for each reader name only for the readers with at
14     least min_classes words and min_instances_per_class instances per word
15
16     Parameters:
17     min_classes (int) (default 2): minimum number of classes per each reader
18     min_instances_per_class (int) (default 26): minimum number of instances
19     per class
20
21     Returns:
22     valid_readers (list of dict): it contains only the readers with at least
23     min_classes words and min_instances_per_class instances per word
24     """
25     # read the json with the words of each reader
26     readers = read_json_file("readers_words.json")
27
28     valid_readers = []
29
30     # for each reader find if it's a valid reader only if he reads C words
31     # for K + Q instances
32     for reader in readers:
33         valid_words = []
34         is_valid_reader = False
35         number_of_valid_words = 0
36
37         # for each word of a reader find if there are for K + Q instances
38         for word in reader['words']:
39             start = []
40             end = []
41             folders = []
42
43             for item in word['folders']:
44                 start += item['start']
45                 end += item['end']
46                 folders += [item['folder']] * len(item['start'])
47
48             new_word = {'word': word['word'], \
49                        'start': start, \
50                        'end': end, \
51                        'folders': folders}
52
53             # if there are at least 26 (default) instances append a new
54             # valid word
55             if (len(new_word['start']) >= min_instances_per_class):
56                 number_of_valid_words += 1
57                 valid_words.append(new_word)
58
59             # if the number of valid words is at least min_classes, then the
60             # reader is valid
61             if (number_of_valid_words >= min_classes):
62                 is_valid_reader = True
63
64             # if the reader is valid then add the reader to the valid readers
65             # list
66             if (is_valid_reader):
67                 valid_readers.append({'reader_name': reader['reader_name'], \
68                                     'words': valid_words})
69
70     return valid_readers

```

### Codice 12: Funzione find\_valid\_readers

La funzione `create_training_validation_test_readers` divide i lettori di training e validation da quelli di test usando la variabile `valid_readers` creata nel codice 12. I primi `number_of_training_readers + number_of_validation_readers` elementi di `valid_readers` sono presi come lettori di training e validation, mentre i restanti come lettori di test.

```
72 def create_training_validation_test_readers(valid_readers ,
73       number_of_training_readers , number_of_test_readers ,
74       number_of_validation_readers):
75     """
76     create_training_validation_test_readers split the valid readers into
77     training+validation readers and test readers
78
79     Parameters:
80     valid_readers (list of dict): list of the valid readers
81     number_of_training_readers (int): size of the training readers
82     number_of_validation_readers (int): size of the validation readers
83     number_of_test_readers (int): size of the test readers
84
85     Returns:
86     training_validation_readers (list of dict): it contains the training and
87     validation readers splitted from valid_readers
88     test_readers (list of dict): it contains the test readers splitted from
89     valid_readers
90     """
91
92     # take the first ("number_of_training_readers" +
93     # number_of_validation_readers") elements of "valid_readers" to create
94     # the training and validation readers
95     training_validation_readers = valid_readers[0 :
96         number_of_training_readers + number_of_validation_readers]
97
98     # take the last "number_of_test_readers" of "valid_readers" to
99     # create the test readers
100     test_readers = valid_readers[number_of_training_readers +
101         number_of_validation_readers:]
102
103     #write_json_file("training_validation_readers.json", training_readers)
104     #write_json_file("test_readers.json", test_readers)
105
106     return training_validation_readers , test_readers
```

### Codice 13: Funzione create\_training\_validation\_test\_readers

La funzione `find_classes` prende in ingresso un lettore e campiona in modo casuale un numero di parole massimo pari a `max_class_number` e un numero di istanze massimo pari a `max_instances_number`. Ritorna un dizionario con chiavi “word”, “start”, “end” e “folders”.

```
100 def find_classes(reader , max_class_number , max_instances_number):
101     """
102     find_classes returns classes, a list of dict that has 'word', 'start',
103     'end', 'folders' keys.
104     'word' value is a string, the word name.
```

```

105     'start', 'end', 'folders' values are lists. The i-th element of 'start'
106     and 'end' value corresponds to the i-th element of 'folder' value.
107     If the reader contains less words than max_class_number then random
108     sample the words, otherwise random sample max_class_number words.
109
110     Parameters:
111     reader (string): one reader from the json file of readers
112     max_class_number (int): maximum class size
113     max_instances_number (int): maximum instance size
114
115     Returns:
116     classes (list): a list of dict that has 'word', 'start',
117     'end', 'folders' keys.
118     """
119     classes = []
120
121     # taking at most C words from the reader
122     # If the reader contains more words than C, random sample C words
123     if (len(reader['words']) >= max_class_number):
124         reader_words = random.sample(reader['words'], max_class_number)
125     # If the reader contains less words than C, random sample the words
126     else:
127         reader_words = random.sample(reader['words'], len(reader['words']))
128
129     for word in reader_words:
130         # numpy.arange returns evenly spaced values within a given interval.
131         # create an array of index to get the start, end and folder of the
132         # same index
133         index_array = list(numpy.arange(len(word['start'])))
134
135         # taking at most max_instances_number instances from each word
136         # If the word contains more instances than max_instances_number,
137         # random sample max_instances_number instances
138         if (len(index_array) >= max_instances_number):
139             index_array = random.sample(index_array, max_instances_number)
140         # If the word contains less instances than max_instances_number,
141         # random sample the index_array
142         else:
143             index_array = random.sample(index_array, len(index_array))
144
145         instance_start = []
146         instance_end = []
147         instance_folder = []
148
149         # sample K instances from every C word class
150         for index in index_array:
151             # get the start, end and folder of the same index
152             instance_start.append(word['start'][index])
153             instance_end.append(word['end'][index])
154             instance_folder.append(word['folders'][index])
155
156         # append the new word of K + Q instances
157         classes.append({'word' : word['word'], \
158                        'start' : instance_start, \
159                        'end' : instance_end, \
160                        'folders' : instance_folder})
161
162     return classes

```

Codice 14: Funzione find\_classes

La funzione `save_dataset` salva gli spettrogrammi delle parole di training e validation usando la funzione `compute_melspectrogram` presentata nel codice 10. Per ottenere i timestamp delle parole e il relativo audio, necessario alla funzione `compute_melspectrogram`, è stato usato `find_classes`, presentato nel codice 14.

Gli spettrogrammi sono salvati in un tensore di dimensione  $K \times 128 \times 51$ , dove  $K$  è il numero di volte in cui la parola è stata ripetuta. Il tensore viene salvato in un file con un nome del tipo “word\_name.pt”, ad esempio “as.pt”, “like.pt”, “the.pt”, ecc.

```

164 def save_dataset(readers, folder_name, dataset_path, audio_file_name,
165                  max_class_number, max_instances_number):
166     """
167     save_dataset saves a pytorch tensor for each word of a reader in a
168     folder named as the reader name.
169
170     Parameters:
171     readers (list of dict): the list of the training and validation readers
172                             saved in training_validation_readers.json
173     folder_name (string): name of the folder in which save the features
174     dataset_path (string): path of the Spoken Wikipedia Corpora dataset
175     audio_file_name (string): name of the ".ogg" audio file
176     max_class_number (int): maximum class size
177     max_instances_number (int): maximum instance size
178
179     Returns:
180     """
181     for reader in tqdm(readers, position = 0):
182         if not (os.path.exists(folder_name + reader['reader_name'])):
183             try:
184                 os.mkdir(folder_name + reader['reader_name'])
185                 classes = find_classes(reader, max_class_number,
186                                     max_instances_number)
187                 for item in tqdm(classes, position = 1, leave = False):
188                     spectrograms = np.empty([0, 128, 51])
189                     for i in range(len(item['start'])):
190                         start_in_sec = item['start'][i]/1000 # conversion
191                             from milliseconds to seconds
192                         end_in_sec = item['end'][i]/1000 # conversion
193                             from milliseconds to seconds
194                         # calculation of the center time of the word
195                         word_center_time = (start_in_sec + end_in_sec)/2
196                         # path of the audio file
197                         audio_file_path = dataset_path + item['folders'][i]
198                             + "/" + audio_file_name
199                         if (os.path.exists(audio_file_path)):
200                             # compute the 128 bit log mel-spectrogram
201                             item_spectrogram = compute_melspectrogram(
202                                 audio_file_path, word_center_time)
203                             # construction of a spectrogram tensor
204                             spectrograms = np.concatenate((spectrograms, [
205                                 item_spectrogram]), axis = 0)
206                     # save the spectrograms tensor only if the first
207                     # dimension is higher than K + Q,
208                     # that is when the word has at least K + Q instances
209                     #if (spectrograms.shape[0] >= K + Q):
210                         # conversion from numpy array to torch.FloatTensor
211                         torch_tensor = torch.FloatTensor(spectrograms)

```

```

203         # save the torch tensor
204         torch.save(torch_tensor, folder_name + reader['
                reader_name'] + "/" + item['word'] + ".pt")
205     except OSError as error:
206         print(error)

```

Codice 15: Funzione save\_dataset

La funzione `save_test_dataset` salva gli spettrogrammi delle parole di test. A differenza del codice 15 vengono salvate al massimo 16 istanze della parola perché durante il test vengono usate solo  $p \in \{1, 5\}$  istanze della parola come *positive set*, mentre tutte le istanze delle altre parole dell'audio compongono il *negative set*. Salvare tutte le istanze della parola risulterebbe dunque superfluo.

```

208 def save_test_dataset(test_readers, reader_paths, test_feature_folder_name,
209                       dataset_path, audio_file_name):
210     """
211     save_test_dataset saves a pytorch tensor for each word of a test reader
212     in a folder named
213     as the audio of the test reader.
214
215     Parameters:
216     test_readers (list of dict): the list of the test readers saved in
217         test_readers.json
218     test_feature_folder_name (string): name of the folder in which save the
219         features
220     dataset_path (string): path of the Spoken Wikipedia Corpora dataset
221     audio_file_name (string): name of the ".ogg" audio file
222
223     Returns:
224     """
225     readers_list = []
226     audio_folders = []
227
228     for reader in test_readers:
229         readers_list.append(reader['reader_name'])
230
231     for reader_name in tqdm(readers_list, position = 0, desc = "test readers
232                             "):
233         for item in reader_paths:
234             if reader_name == item['reader_name']:
235                 audio_folders += item['folder']
236
237     for audio_folder in tqdm(audio_folders, position = 0, desc = "audio
238                             folders"):
239         if not (os.path.exists(test_feature_folder_name + audio_folder)):
240             try:
241                 os.mkdir(test_feature_folder_name + audio_folder)
242                 path = dataset_path + audio_folder + "/target_words.json"
243                 words = read_json_file(path)
244                 for word in words:
245                     spectrograms = np.empty([0, 128, 51])
246                     if (len(word['start']) > 16):
247                         indices = random.sample(list(numpy.arange(len(word['
248                             start']))), 16)
249                     else:
250                         indices = random.sample(list(numpy.arange(len(word['
251                             start']))), len(word['start']))

```

```

244         for i in indices:
245             start_in_sec = word['start'][i]/1000 # conversion
                from milliseconds to seconds
246             end_in_sec = word['end'][i]/1000      # conversion
                from milliseconds to seconds
247             # calculation of the center time of the word
248             word_center_time = (start_in_sec + end_in_sec)/2
249             # path of the audio file
250             audio_file_path = dataset_path + audio_folder +
                audio_file_name
251             if (os.path.exists(audio_file_path)):
252                 # compute the 128 bit log mel-spectrogram
253                 item_spectrogram = compute_melspectrogram(
                    audio_file_path, word_center_time)
254                 # construction of a spectrogram tensor
255                 spectrograms = np.concatenate((spectrograms, [
                    item_spectrogram]), axis = 0)
256             # save the spectrograms tensor only if the first
                dimension is higher than K + Q,
257             # that is when the word has at least K + Q instances
258             #if (spectrograms.shape[0] >= K + Q):
259                 # conversion from numpy array to torch.FloatTensor
260             torch_tensor = torch.FloatTensor(spectrograms)
261             # save the torch tensor
262             torch.save(torch_tensor, test_feature_folder_name +
                audio_folder + "/" + word['word'] + ".pt")
263
264     except OSError as error:
265         print(error)

```

Codice 16: Funzione save\_test\_dataset

## 7 Implementazione della Prototypical network

### 7.1 Definizione della rete

La CNN che funge da rete di embedding per la Prototypical Network è costituita da 4 strati convoluzionali, come si può vedere nel codice 17. Il primo di essi ha come ingresso un singolo canale e come uscita 64 mentre i tre successivi traspongono 64 canali in altri 64. Ogni blocco convoluzionale ha 4 fasi:

- `nn.Conv2d(in_channels, out_channels, 3, padding=1)`: Convoluzione bidimensionale con un kernel  $3 \times 3$  i cui parametri vengono aggiornati ad ogni backward. Viene effettuato un padding di zeri ai bordi dell'ingresso.
- `nn.BatchNorm2d(out_channels)`: La batch normalization è un metodo utilizzato per rendere le reti neurali artificiali più veloci e stabili attraverso la normalizzazione degli input dei livelli con re-centering and re-scaling.
- `nn.ReLU()`: il rettificatore è una funzione di attivazione definita come la parte positiva del suo argomento.  $f(x) = \max(0, x)$
- `nn.MaxPool2d(2)`: Il max-pooling è un metodo per ridurre la dimensione di un'immagine, suddividendola in blocchi e tenendo solo quello col valore più alto.

Al termine dei blocchi viene effettuato il reshape dell'uscita schiacciandola in una sola dimensione, `x.view(x.size(0), -1)`, in modo da avere una matrice le cui righe rappresentano gli embedding vettoriali.

La funzione `count_parameters` ritorna il numero di parametri della rete che, in accordo con [6], devono essere compresi tra 120000 e 230000.

```
1 import torch.nn as nn
2
3 def count_parameters(model):
4     return sum(p.numel() for p in model.parameters() if p.requires_grad)
5
6 def conv_block(in_channels, out_channels):
7
8     return nn.Sequential(
9         nn.Conv2d(in_channels, out_channels, 3, padding=1),
10        nn.BatchNorm2d(out_channels),
11        nn.ReLU(),
12        nn.MaxPool2d(2)
13    )
14
15 class Protonet(nn.Module):
16     def __init__(self):
```



```

17         super(Protonet, self).__init__()
18         self.encoder = nn.Sequential(
19             conv_block(1, 64),
20             conv_block(64, 64),
21             conv_block(64, 64),
22             conv_block(64, 64)
23         )
24     def forward(self, x):
25         (num_samples, mel_bins, seq_len) = x.shape
26         # tensor 3D to tensor 4D with 3rd dimension = 1
27         x = x.view(-1, 1, mel_bins, seq_len)
28         x = self.encoder(x)
29         return x.view(x.size(0), -1)

```

Codice 17: protonet.py

## 7.2 Training

Il primo for rappresenta l'inizio degli episodi. Il numero totale di iterazioni è 60000 come in [6]; per ogni episodio viene richiamata la funzione `batch_sample`.

Con `loss_out.backward()` viene utilizzata la loss per la retropropagazione che aggiorna i parametri della rete.

```

1  from tqdm import trange, tqdm
2  import torch
3  import numpy as np
4  import scipy
5  from scipy import io
6
7
8  if torch.cuda.is_available():
9      device = torch.device("cuda")
10     print("Device: {}".format(device))
11     print("Device name: {}".format(torch.cuda.get_device_properties(device).name))
12 else:
13     device = torch.device("cpu")
14
15 C = 10 # classes
16 K = 5 # instances per class
17 Q = 16 # query set size
18
19 model = Protonet()
20 if torch.cuda.is_available():
21     model.to(device='cuda')
22
23 print("Model parameters: {}".format(count_parameters(model)))
24
25 optim = torch.optim.Adam(model.parameters(), lr = 0.001)
26
27 training_readers, validation_readers = get_training_validation_readers("/
    Training_validation_features/", C)
28
29 print ("Training...")
30
31 last_accuracy = 0.0

```

```

32 train_loss = []
33 train_acc = []
34
35
36 # To construct a C-way K-shot training episode, we randomly sample a reader
37 # from the training set, sample C word classes from the reader, and sample K
38 # instances per class as the support set.
39
40 for episode in trange(60000, desc = "episode", position = 0, leave = True):
41     query, support = batch_sample(training_readers, C, K, Q)
42     if torch.cuda.is_available():
43         support = support.to(device='cuda')
44         query = query.to(device='cuda')
45
46     model.train()
47     optim.zero_grad()
48     loss_out, acc_val = loss(support, query, model)
49
50     loss_out.backward()
51     optim.step()
52
53     train_loss.append(loss_out.item())
54     train_acc.append(acc_val.item())
55
56     if (episode+1)%5000 == 0:
57         valid_loss = []
58         valid_acc = []
59         print ("\nValidation...")
60
61         model.eval()
62
63         for validation_episode in range(1000):
64             query, support = batch_sample(validation_readers, C, K, Q)
65             if torch.cuda.is_available():
66                 support = support.to(device='cuda')
67                 query = query.to(device='cuda')
68
69             val_loss, acc_val = loss(support, query, model)
70             optim.step()
71
72             valid_loss.append(val_loss.item())
73             valid_acc.append(acc_val.item())
74
75         print ("\nValidation accuracy: {}".format(np.mean(valid_acc)))
76
77         if np.mean(valid_acc) > last_accuracy:
78
79             torch.save({
80                 'epoch': episode+1,
81                 'model_state_dict': model.state_dict(),
82                 'optimizer_state_dict': optim.state_dict(),
83                 'train_loss': train_loss,
84                 'train_acc': train_acc,
85                 'valid_loss': valid_loss,
86                 'valid_acc': valid_acc,
87                 'avg_loss_tr': np.mean(train_loss),
88                 'avg_acc_tr': np.mean(train_acc),
89                 'avg_loss_val': np.mean(valid_loss),
90                 'avg_acc_val': np.mean(valid_acc),
91             }, "/Modelli-Prototypical-Network/prototypical_model_C{}_K{}.
92                 pt".format(C, K))

```

```

93     last_accuracy = np.mean(valid_acc)
94
95     scipy.io.savemat('/Modelli-Prototypical-Network/
        prototypical_results_C{}_K{}.mat'.format(C, K), {'train_loss':
        train_loss, 'train_acc': train_acc, 'valid_loss': valid_loss,
        'valid_acc': valid_acc})

```

Codice 18: protonet\_training.py

La funzione `get_training_validation_readers`, a partire dalla lista di lettori di training/validation, seleziona quelli che hanno almeno un numero di parole diverse pari a  $C$  e li suddivide tra training e validation seguendo il rapporto usato in [6]. Per fare ciò scorre l'elenco delle directory dei lettori e salva su un array le parole a loro associate. Al termine di questa operazione, se il numero di parole è almeno  $C$  il lettore è ritenuto valido e il suo nome viene aggiunto a una lista. Infine la lista viene scissa in lettori di training e di validation.

```

1  def get_training_validation_readers(features_folder, C):
2      """
3      get_training_validation_readers returns training and validation readers.
4      From the training and validation readers, it takes only the readers with
5      at least C words and split the list in training readers and validation
6      readers.
7
8      Parameters:
9      features_folder (list of string): list of the path of the training and
        validation
10     C (int): number of classes
11
12     Returns:
13     training_readers (list of string): list of the training readers paths
14     validation_readers (list of string): list of the validation readers paths
15     """
16     readers_path = []
17     train_val_readers = []
18     # scan each reader folder in the feature_folder
19     for entry in os.scandir(features_folder):
20         # create a list of reader names
21         readers_path.append(entry.path)
22     for reader_path in readers_path:
23         words = []
24         for word in os.scandir(reader_path):
25             # create a list containing the path of the words
26             words.append(word.path)
27         if (len(words) >= C):
28             train_val_readers.append(reader_path)
29     train_val_readers = random.sample(train_val_readers, len(train_val_readers))
30     training_readers = train_val_readers[:int(138/153*len(train_val_readers))]
31     validation_readers = train_val_readers[int(138/153*len(train_val_readers)):]
32     return training_readers, validation_readers

```

Codice 19: get\_training\_validation\_readers

La funzione `batch_sample` campiona a caso uno dei lettori di training e di questo seleziona casualmente  $C$  parole. Per ognuna di queste parole si ricava il numero di istanze presenti del dataset e vengono campionate  $K + Q$  istanze random della parola. Le  $K + Q$  istanze vengono divise in  $K$  istanze del support set e  $Q$  stanze del query set. La funzione ritorna due tensori `query` e `support`, rispettivamente di dimensioni  $C \times Q \times 128 \times 51$  e  $C \times K \times 128 \times 51$ .

```

1  import os
2  import numpy
3  import random
4
5  def batch_sample(features, C, K, Q = 16):
6      """
7      batch_sample returns the support and query set.
8      It reads each folder in feature_folder and load the tensor with the
9      spectrograms of the word.
10     Then random sample the instances (spectrograms) of the word to get only
11     K+Q spectrograms.
12     The first K spectrograms compose the support set and the last Q ones
13     compose the query set.
14
15     Parameters:
16     features (list): training/validation features paths
17     C (int): class size
18     K (int): support set size
19     Q (int): query set size (default: 16)
20
21     Returns:
22     support (torch.FloatTensor): support set
23     query (torch.FloatTensor): query set
24     """
25     # initialize support tensor of dimension 0 x K x 128 x 51
26     support = torch.empty([0, K, 128, 51])
27     # initialize query tensor of dimension 0 x Q x 128 x 51
28     query = torch.empty([0, Q, 128, 51])
29     # random sample a reader
30     reader = random.sample(features, 1)[0]
31     words = []
32     # scan the torch tensor saved in each reader folder
33     for word in os.scandir(reader):
34         # create a list containing the path of the words
35         words.append(word.path)
36     # random sample C paths of the words of a reader
37     words = random.sample(words, C)
38     # randomize the instances of each word
39     for word in words:
40         # load the tensor containing the spectrograms of the instances of
41         # one word
42         spectrogram_buf = torch.load(word)
43         # get the spectrogram tensor shape
44         x_dim, y_dim, z_dim = spectrogram_buf.shape
45         # get the number of instances
46         instances_number = (spectrogram_buf.shape)[0]
47         # random sample K + Q indices
48         index = random.sample(list(torch.arange(instances_number)), K + Q)
49         # initialize the spectrogram tensor
50         spectrogram = torch.empty([0, 128, 51])
51         for i in index:
52             # concatenate spectrogram_buf with spectrogram to get a new

```

```

50         # tensor of random sampled instances of the word
51         spectrogram = torch.cat((spectrogram, (spectrogram_buf[i, :, :])
52                                   .view(1, y_dim, z_dim)), axis = 0)
52         # concatenate the first K spectrograms with the support set
53         support = torch.cat((support, (spectrogram[:K]).view(1, K, y_dim,
54                                   z_dim)), axis = 0)
54         # concatenate the last Q spectrograms with the query set
55         query = torch.cat((query, (spectrogram[K:K+Q]).view(1, Q, y_dim,
56                                   z_dim)), axis = 0)
56     return query, support

```

Codice 20: batch\_sample

Una volta ottenute, le feature del support set e query set vengono date in ingresso alla funzione `loss`, mostrata nel codice 21. Entrambi i set vengono ridimensionati, rispettivamente da  $C \times K \times 128 \times 51$  a  $(C \cdot K) \times 128 \times 51$  e da  $C \times Q \times 128 \times 51$  a  $(C \cdot Q) \times 128 \times 51$ . Vengono poi concatenate, risultando in un tensore  $(C \cdot (Q + K)) \times 128 \times 51$ , per poter essere passate al modello che calcolerà gli embedding per ogni istanza. Gli elementi del support set che fanno parte della stessa classe andranno a costituire i prototipi tramite una media dei loro valori.

Il parametro `target_inds` viene costruito ripetendo l'indice di ogni classe,  $0, 1, \dots, C$ , per il numero di query, 16, e sarà utilizzato per poter ricavare a che classe appartiene ogni query.

Vengono successivamente calcolate le distanze rispetto ai prototipi degli embedding di ogni classe tramite la funzione `euclidean_dist` che ritorna una matrice di dimensioni  $(C \cdot Q) \times C$ . L' $i$ -esima riga e la  $j$ -esima colonna rappresentano la distanza euclidea tra gli embedding della  $i$ -esima query ( $i = 1, \dots, C \cdot Q$ ) e quelli del  $j$ -esimo ( $j = 1, \dots, C$ ) prototipo.

Per ogni query viene poi usata la funzione di attivazione softmax, definita in (2), i cui argomenti sono le distanze tra la query e i prototipi delle varie classi. La funzione loss da minimizzare è il logaritmo del softmax preso con segno negativo, come nell'equazione (3). Se la query è vicina al prototipo, la funzione softmax tende a 1 e di conseguenza il suo logaritmo a 0, dunque la loss tende a 0, come dovrebbe appunto risultare. Nel caso in cui, invece, la query è molto lontana dal prototipo, la funzione softmax tende a 0 e il suo logaritmo a  $-\infty$ . In questo caso, la loss, definita come il logaritmo del softmax preso con segno negativo, tende a  $+\infty$ , come ci si aspetta.

Poiché la funzione `log_softmax` di Pytorch ritorna una matrice  $(C \cdot Q) \times C$ , per ogni riga occorre prendere solo l'elemento relativo alla classe del prototipo. Questo è possibile utilizzando la funzione `gather` di Pytorch.

L'accuracy invece è determinata dal numero dei casi in cui il prototipo a distanza minima da una query (o il massimo del logaritmo del softmax) corrisponde al prototipo della classe associata alla query.

```

1 import torch

```

```

2 import torch.nn.functional as F
3
4 def euclidean_dist(x, y):
5     """
6     euclidean_dist computes the euclidean distance from two arrays x and y
7
8     Parameters:
9     x (torch.FloatTensor): query array
10    y (torch.FloatTensor): prototype array
11
12    Returns:
13    torch.pow(x - y, 2).sum(2) (double): the euclidean distance from x and y
14    """
15    # x: N x D
16    # y: M x D
17    n = x.size(0)
18    m = y.size(0)
19    d = x.size(1)
20    assert d == y.size(1)
21
22    x = x.unsqueeze(1).expand(n, m, d)
23    y = y.unsqueeze(0).expand(n, m, d)
24
25    return torch.pow(x - y, 2).sum(2)
26
27 def loss(xs, xq, model):
28     """
29     loss returns the loss and accuracy value. It calculates p_y, the loss
30     softmax over distances to the prototypes in the embedding space. We need
31     to minimize the negative log-probability of p_y to proceed the learning
32     process.
33
34     Parameters:
35     xs (torch.FloatTensor): support set
36     xq (torch.FloatTensor): query set
37     model (torch.nn.Module): neural network model
38
39     Returns:
40     loss_val (double): loss value
41     acc_val (double): accuracy value
42     """
43     # xs = Variable()
44     n_class = xs.size(0)
45     assert xq.size(0) == n_class
46     n_support = xs.size(1)
47     n_query = xq.size(1)
48
49     target_inds = torch.arange(0, n_class).view(n_class, 1, 1).expand(
50         n_class, n_query, 1).long()
51
52     if torch.cuda.is_available():
53         target_inds = target_inds.to(device='cuda')
54
55     x = torch.cat([xs.view(n_class * n_support, *xs.size()[2:]),
56                    xq.view(n_class * n_query, *xq.size()[2:]), 0])
57
58     embeddings = model(x)
59
60     embeddings_dim = embeddings.size(-1)
61
62     prototypes = embeddings[:n_class*n_support].view(n_class, n_support,
63                                                         embeddings_dim).mean(1)

```

```

62
63     queries = embeddings[n_class*n_support:]
64
65     dists = euclidean_dist(queries, prototypes)
66
67     log_p_y = F.log_softmax(-dists, dim=1).view(n_class, n_query, -1)
68
69     loss_val = -log_p_y.gather(2, target_inds).squeeze().view(-1).mean()
70
71     _, y_hat = log_p_y.max(2)
72     acc_val = torch.eq(y_hat, target_inds.squeeze()).float().mean()
73
74     return loss_val, acc_val

```

Codice 21: protonet\_loss.py

## 7.3 Validation

Ogni 5000 episodi del training vengono effettuati 1000 episodi di validation. Come nel training vengono ricavati support set e query set, ma in questo caso dai lettori di validation. Per questi vengono calcolati con la funzione `loss` allo stesso modo `loss` e `accuracy` e vengono salvati. In questo caso però i parametri del modello non vengono aggiornati. Questo processo è necessario per verificare l'effettività del modello su ingressi mai visti in fase di training. Al termine dei 1000 episodi di validation, se la media dell'accuracy è migliore di quella calcolata nel passo precedente il modello viene salvato e sovrascritto, altrimenti viene ignorato. In questo modo si salva il modello migliore durante il training, cercando di evitare sia `underfitting` sia `overfitting`.

## 7.4 Test

Per eseguire il test, l'idea è quella di costruire un set positivo e un set negativo con i quali confrontare la query, come mostrato in 1. Mentre il set positivo è costituito da istanze della stessa parola, quello negativo comprende vari campioni di parole diverse e rappresenta un generico esempio di tutto ciò che non comprende la parola da cercare.

Nella fase di test, seguendo l'articolo di riferimento, l'oggetto di interesse non sono i lettori come nel caso di training e validation. Un episodio di test è formato da un audio di un lettore di test. Da questi sarà necessario identificare un gruppo di istanze di una parola, il positive set e, campionando nell'audio, si potrà verificare se la rete è in grado di riconoscere la parola designata.

Per ogni audio quindi viene ricavato il numero di parole e le relative istanze. Per ognuna di esse viene ricavato un positive set e un negative set con i quali è possibile verificare la rete. Questo processo viene ripetuto 10

volte come in [6], in modo da generalizzare il più possibile il test. Infatti, il positive set è ricavato campionando casualmente  $p$  istanze della parola che si sta analizzando, mentre il negative set è ottenuto campionando  $n$  istanze fra tutte le altre parole dello stesso audio.

Le predizioni sono contenute nel vettore `y_pred`, mentre in `y_true` sono presenti le label reali di ogni query.

La funzione `precision_recall_curve` utilizza `y_pred` e `y_true` per calcolare precision e recall. Infine, `sklearn.metrics.auc` calcola la AUPRC. Questo calcolo avviene al termine delle operazioni per ogni audio. Una volta terminati gli audio viene calcolata media e varianza degli AUPRC.

```

77 def main():
78     p = 5
79     n = 10
80
81     C = 2
82     K = 1
83
84     model = Protonet()
85     optim = torch.optim.Adam(model.parameters(), lr = 0.001)
86
87     if torch.cuda.is_available():
88         checkpoint = torch.load("Models/Prototypical/prototypical_model_C{
            _K{}.pt".format(C, K), map_location=torch.device('cuda'))
89     else:
90         checkpoint = torch.load("Models/Prototypical/prototypical_model_C{
            _K{}.pt".format(C, K), map_location=torch.device('cpu'))
91     model.load_state_dict(checkpoint['model_state_dict'])
92     optim.load_state_dict(checkpoint['optimizer_state_dict'])
93
94
95     model.eval()
96
97     if torch.cuda.is_available():
98         model.to(device='cuda')
99
100     auc_list = []
101     for audio in tqdm(os.listdir("Test_features/"), desc = "Test features"):
102         y_pred = []
103         y_true = []
104         # getting the number of target keywords in each audio
105         target_keywords_number = len([name for name in os.listdir(audio) if
            os.path.isfile(os.path.join(audio, name))])
106
107         for i in range(target_keywords_number):
108             for j in range(10):
109                 negative_set, positive_set, query_set =
                    get_negative_positive_query_set(p, n, i, audio)
110
111                 if torch.cuda.is_available:
112                     negative_set = negative_set.to(device='cuda')
113                     positive_set = positive_set.to(device='cuda')
114                     query_set = query_set.to(device='cuda')
115
116                 y_pred_tmp, y_true_tmp = test_predictions(model,
                    negative_set, positive_set, query_set, n, p)
117                 y_pred_tmp = y_pred_tmp.cpu().tolist()

```



```

118         y_true_tmp = y_true_tmp.cpu().tolist()
119
120         y_pred.extend(y_pred_tmp)
121         y_true.extend(y_true_tmp)
122
123         precision, recall, thresholds = sklearn.metrics.
            precision_recall_curve(y_true, y_pred)
124         auc_tmp = sklearn.metrics.auc(recall, precision)
125         auc_list.append(auc_tmp)
126
127     auc = np.mean(auc_list)
128     print("Area under precision recall curve: {}".format(auc))
129     auc_std_dev = np.std(auc_list)
130     print("Standard deviation area under precision recall curve: {}".format(
        auc_std_dev))

```

Codice 22: protonet\_test.py

Per costruire il positive e negative set viene usata la funzione `get_negative_positive_query_set`. Tramite un indice che indica le istanze della parola appartenente alla classe positiva vengono campionati  $p$  spettrogrammi che vanno a costituire il positive set. Le restanti istanze della stessa parola vengono inserite invece nel query set come esempio di spettrogramma che la rete dovrebbe riconoscere. Tra l'elenco delle parole viene quindi rimossa la parola positiva con `words.remove(pos_word)`. Usando le restanti vengono ricavate  $n$  esempi di negative set campionando a caso una delle parole e una delle istanze di essa. Allo stesso modo viene formata la seconda metà del query set che rappresenta gli esempi negativi. Il numero di query negative viene preso uguale a quello delle positive.

```

33 def get_negative_positive_query_set(p, n, i, audio):
34     # initialize support tensor of dimension p x 128 x 51
35     positive_set = torch.empty([0, 128, 51])
36     negative_set = torch.empty([0, 128, 51])
37     # initialize query tensor of dimension n x 128 x 51
38     query_set = torch.empty([0, 128, 51])
39
40     words = []
41     for word in os.scandir(audio.path):
42         words.append(word.path)
43
44     pos_word = words[i]
45
46     spectrograms = torch.load(pos_word)
47     index = np.arange(spectrograms.shape[0])
48     pos_index = random.sample(list(index), p)
49     for i in pos_index:
50         pos = spectrograms[i, :, :]
51         positive_set = torch.cat((positive_set, pos.view(1, 128, 51)), axis=0)
52     for i in index:
53         if i not in pos_index:
54             query = spectrograms[i, :, :]
55             query_set = torch.cat((query_set, query.view(1, 128, 51)), axis=0)
56

```

```

57     words.remove(pos_word)
58
59     for i in range(n):
60         neg = random.sample(words, 1)
61         spectrograms = torch.load(neg[0])
62         index = np.arange(spectrograms.shape[0])
63         neg_index = random.sample(list(index), 1)
64         neg = spectrograms[neg_index, :, :]
65         negative_set = torch.cat((negative_set, neg.view(1, 128, 51)), axis
66                                 = 0)
67
68     for i in range(query_set.shape[0]):
69         query_sample = random.sample(words, 1)
70         spectrograms = torch.load(query_sample[0])
71         index = np.arange(spectrograms.shape[0])
72         query_index = random.sample(list(index), 1)
73         query = spectrograms[query_index, :, :]
74         query_set = torch.cat((query_set, query.view(1, 128, 51)), axis = 0)
75
76     return negative_set, positive_set, query_set

```

Codice 23: Funzione `get_negative_positive_query_set`

A questo punto, i tre set e il modello sono passati alla funzione `test_predictions`. Allo stesso modo di training e validation vengono calcolati gli embedding per ogni istanza e i prototipi delle due classi: positive set e negative set. Vengono calcolate le distanze che sono passate con segno negativo alla funzione `log_softmax`. Tra i risultati viene selezionato l'elemento 0 della terza dimensione che rappresenta la probabilità che la query appartenga alla classe positiva.

```

11 def test_predictions(model, negative_set, positive_set, query_set, n, p):
12     n_class = 2
13     n_query = query_set.size(0)
14
15     xs = torch.cat((positive_set, negative_set), 0)
16     x = torch.cat((xs, query_set), 0)
17
18     embeddings = model(x)
19     embeddings_dim = embeddings.size(-1)
20
21     positive_embeddings = embeddings[:p].view(1, p, embeddings_dim).mean(1)
22     negative_embeddings = embeddings[p:p+n].view(1, n, embeddings_dim).mean
23     (1)
24     pos_neg_embeddings = torch.cat((positive_embeddings,
25                                     negative_embeddings), 0)
26     query_embeddings = embeddings[p+n:]
27     dists = euclidean_dist(query_embeddings, pos_neg_embeddings)
28
29     target_inds = torch.arange(n_class-1, -1, step = -1).view(n_class, 1).
30     expand(n_class, int(n_query/2)).long()
31
32     p_y = F.softmax(-dists, dim = 1).view(n_class, int(n_query/2), -1)
33
34     return p_y[:, :, 0].view(-1).detach(), target_inds.reshape(1, -1).squeeze
35     ()

```

Codice 24: Funzione `test_prediction`

## 8 Implementazione della Relation network

### 8.1 Definizione della rete

Le reti neurali necessarie per la relation network sono effettivamente due.

La prima, chiamata `CNNEncoder` ha lo stesso scopo della rete nella prototypical, ovvero a partire dalle feature in ingresso produce degli embedding che possono essere confrontati. Anche in questo caso i layer sono 4 e sono costituiti da:

- `nn.Conv2d()`: Convoluzione bidimensionale con un kernel  $3 \times 3$  i cui parametri vengono aggiornati ad ogni backward. Viene effettuato un padding di zeri ai bordi dell'ingresso.
- `nn.BatchNorm2d()`: La batch normalization è un metodo utilizzato per rendere le reti neurali artificiali più veloci e stabili attraverso la normalizzazione degli input dei livelli con re-centering and re-scaling.
- `nn.ReLU()`: il rettificatore è una funzione di attivazione definita come la parte positiva del suo argomento.  $f(x) = \max(0, x)$
- `nn.MaxPool2d(2)`: Il max-pooling è un metodo per ridurre la dimensione di un'immagine, suddividendola in blocchi e tenendo solo quello col valore più alto.

Seguendo le indicazioni in [4] i primi due blocchi hanno padding pari a 0 mentre negli ultimi due padding=1. Inoltre, per soddisfare le dimensioni necessarie in ingresso alla seconda rete neurale, il Max Pooling è presente solo nei primi 3 layer. Questo concetto sarà approfondito nel capitolo di training.

La seconda rete è detta `RelationNetwork` e ha lo scopo di confrontare le concatenazioni di embedding per predire o meno la loro somiglianza. Essa è costituita da 2 layer di `nn.Conv2d()`, `nn.BatchNorm2d()`, `nn.ReLU()` e `nn.MaxPool2d(2)` al termine la funzione `out.view(out.size(0), -1)` ritorna una matrice le cui righe rappresentano gli embedding vettoriali. In seguito vengono applicati una ulteriore relu e una sigmoide.

```
1 import torch
2 import torch.nn as nn
3 import math
4 import torch.nn.functional as F
5
6 class CNNEncoder(nn.Module):
7     """
8     CNNEncoder is the embedding module. The architecture consists of 4
9     convolutional block contains a 64-filter 3 X 3 convolution, a batch
10    normalization and a ReLU nonlinearity layer respectively.
```

```

11 The first 3 blocks also contain a 2 X 2 max-pooling layer while the last
12 two do not. We do so because we need the output feature maps for further
13 convolutional layers in the relation module
14 """
15 def __init__(self):
16     super(CNNEncoder, self).__init__()
17     self.layer1 = nn.Sequential(
18         nn.Conv2d(1,64,kernel_size=3,padding=0),
19         nn.BatchNorm2d(64, momentum=1, affine=True),
20         nn.ReLU(),
21         nn.MaxPool2d(2))
22     self.layer2 = nn.Sequential(
23         nn.Conv2d(64,64,kernel_size=3,padding=0),
24         nn.BatchNorm2d(64, momentum=1, affine=True),
25         nn.ReLU(),
26         nn.MaxPool2d(2))
27     self.layer3 = nn.Sequential(
28         nn.Conv2d(64,64,kernel_size=3,padding=1),
29         nn.BatchNorm2d(64, momentum=1, affine=True),
30         nn.ReLU(),
31         nn.MaxPool2d(2))
32     self.layer4 = nn.Sequential(
33         nn.Conv2d(64,64,kernel_size=3,padding=1),
34         nn.BatchNorm2d(64, momentum=1, affine=True),
35         nn.ReLU())
36         #nn.MaxPool2d(2))
37
38     def forward(self,x):
39         out = self.layer1(x)
40         out = self.layer2(out)
41         out = self.layer3(out)
42         out = self.layer4(out)
43         #out = out.view(out.size(0),-1)
44         return out # 64
45
46 class RelationNetwork(nn.Module):
47     """
48     The RelationNetwork is the relation module. It consists of two
49     convolutional blocks and two fully-connected layers. Each of
50     convolutional block is a 3 X 3 convolution with 64 filters followed
51     by batch normalization, ReLU non-linearity and 2 X 2 max-pooling.
52     The two fully-connected layers are 8 and 1 dimensional, respectively.
53     All fully-connected layers are ReLU except the output layer is Sigmoid
54     in order to generate relation scores in a reasonable range for all
55     versions of our network architecture.
56     """
57     def __init__(self,input_size,hidden_size):
58         super(RelationNetwork, self).__init__()
59         self.layer1 = nn.Sequential(
60             nn.Conv2d(128,64,kernel_size=3,padding=1),
61             nn.BatchNorm2d(64, momentum=1, affine=True),
62             nn.ReLU(),
63             nn.MaxPool2d(2))
64         self.layer2 = nn.Sequential(
65             nn.Conv2d(64,64,kernel_size=3,padding=1),
66             nn.BatchNorm2d(64, momentum=1, affine=True),
67             nn.ReLU(),
68             nn.MaxPool2d(2))
69         self.fc1 = nn.Linear(input_size, hidden_size)
70         self.fc2 = nn.Linear(hidden_size, 1)
71
72     def forward(self,x):

```

```

73     out = self.layer1(x) # (CLASS_NUM * BATCH_NUM_PER_CLASS *
    CLASS_NUM) X FEATURE_DIM X 2 X 2
74     out = self.layer2(out) # (CLASS_NUM * BATCH_NUM_PER_CLASS *
    CLASS_NUM) X FEATURE_DIM X 1 X 1
75     out = out.view(out.size(0), -1) # (CLASS_NUM * BATCH_NUM_PER_CLASS
    * CLASS_NUM) X FEATURE_DIM
76     out = F.relu(self.fc1(out)) # (CLASS_NUM * BATCH_NUM_PER_CLASS
    * CLASS_NUM) X RELATION_DIM
77     #out = F.sigmoid(self.fc2(out)) # deprecated
78     out = torch.sigmoid(self.fc2(out)) # (CLASS_NUM *
    BATCH_NUM_PER_CLASS * CLASS_NUM) X 1
79     return out
80
81 def weights_init(m):
82     classname = m.__class__.__name__
83     if classname.find('Conv') != -1:
84         n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
85         m.weight.data.normal_(0, math.sqrt(2. / n))
86         if m.bias is not None:
87             m.bias.data.zero_()
88     elif classname.find('BatchNorm') != -1:
89         m.weight.data.fill_(1)
90         m.bias.data.zero_()
91     elif classname.find('Linear') != -1:
92         n = m.weight.size(1)
93         m.weight.data.normal_(0, 0.01)
94         m.bias.data = torch.ones(m.bias.data.size())

```

Codice 25: relation\_network.py

## 8.2 Training

Come nella rete precedente, il primo passo consiste nel campionare i lettori di training e validation con la funzione `get_training_validation_readers`. Le due reti `CNNEncoder` e `RelationNetwork` con i rispettivi pesi vengono inizializzate. Per ogni episodio viene poi campionato un support set e query set per un lettore designato. Le istanze complessive vengono poi passate alla prima rete per costruire gli embedding. Gli embedding del support set hanno ora dimensione  $(C \cdot K) \times 64 \times 15 \times 5$  mentre quelli del query set sono  $(C \cdot Q) \times 64 \times 15 \times 5$ .

Prima di essere passati alla seconda rete questi tensori necessitano di avere le ultime due dimensioni pari a 5. In questo modo, attraversando due volte lo strato di max pooling passeranno da  $5 \times 5$  a  $2 \times 2$  e infine a  $1 \times 1$ . In seguito al flattening le ultime due dimensioni scompaiono e l'ingresso alla relu ha la dimensione corretta:  $(C \cdot Q \cdot C) \times 64$ . Le concatenazioni hanno dimensione  $(C \cdot Q \cdot C) \times 64 \times 5 \times 5$ , infatti ogni query  $(C \times Q)$  va concatenata con il vettore che rappresenta ognuna delle classi ( $C$ ). Quest'ultimo vettore è ottenuto sommando elemento per elemento gli embedding delle istanze di ogni classe usando `torch.sum(sample_features, 1).squeeze(1)`. La rete decisionale restituisce quindi un tensore `relations`  $(C \cdot K \cdot C) \times 1 \times 1$  che

viene poi suddiviso in una matrice  $(C \cdot Q) \times C$  in cui in ogni riga sono presenti le probabilità che una certa query appartenga ad una determinata classe. La matrice `one_hot_labels` ha la stessa dimensione di `relations` ma è costituito da zeri tranne nelle colonne in cui la classe corrisponde realmente alla query, in cui il valore è pari a 1. Questa rappresenta il risultato ideale che vogliamo in uscita dalla rete. Usando quindi la funzione `mse` con parametri `relations` e `one_hot_labels` otteniamo la loss, calcolata come mean square error, che viene utilizzata per aggiornare i parametri delle due reti con `loss.backward()`.

```

1  import torch
2  import torch.nn as nn
3  from torch.autograd import Variable
4  import torch.nn.functional as F
5  from tqdm import trange, tqdm
6  import numpy as np
7  import scipy
8  from scipy import io
9
10 # Hyper Parameters
11 FEATURE_DIM = 64
12 RELATION_DIM = 8
13 CLASS_NUM = 5
14 SAMPLE_NUM_PER_CLASS = 1
15 BATCH_NUM_PER_CLASS = 16
16 EPISODE = 60000
17 TEST_EPISODE = 1000
18 LEARNING_RATE = 0.001
19
20 if torch.cuda.is_available():
21     device = torch.device("cuda")
22     print("Device: {}".format(device))
23     print("Device name: {}".format(torch.cuda.get_device_properties(device).
24                                     name))
25 else:
26     device = torch.device("cpu")
27
28 training_readers, validation_readers = get_training_validation_readers("/
29     Training_validation_features/", CLASS_NUM)
30
31 # init neural network
32 feature_encoder = CNNEncoder()
33 relation_network = RelationNetwork(FEATURE_DIM, RELATION_DIM)
34
35 feature_encoder.apply(weights_init)
36 relation_network.apply(weights_init)
37
38 if torch.cuda.is_available():
39     feature_encoder.to(device='cuda')
40     relation_network.to(device='cuda')
41
42 feature_encoder_optim = torch.optim.Adam(feature_encoder.parameters(), lr =
43     LEARNING_RATE)
44 relation_network_optim = torch.optim.Adam(relation_network.parameters(), lr
45     = LEARNING_RATE)
46
47 print("Training...")
48

```

```

45 train_loss = []
46 validation_loss = []
47
48 last_accuracy = 0.0
49
50 for episode in tqdm(range(EPISODE), desc = "training episode", position=0):
51
52     # sample datas
53     batches, samples = batch_sample(training_readers, CLASS_NUM,
54                                     SAMPLE_NUM_PER_CLASS, BATCH_NUM_PER_CLASS) # samples: C X K X 128 X
55                                     51, batches: C X Q X 128 X 51
56
57     samples = samples.view(CLASS_NUM * SAMPLE_NUM_PER_CLASS, 1, *samples.
58                           size()[2:]) # (C X K) X 1 X 51 X 51
59     batches = batches.view(CLASS_NUM * BATCH_NUM_PER_CLASS, 1, *batches.size
60                           ()[2:]) # (C X Q) X 1 X 51 X 51
61
62     if torch.cuda.is_available():
63         samples = samples.to(device='cuda')
64         batches = batches.to(device='cuda')
65
66     # calculate features
67     sample_features = feature_encoder(Variable(samples)) # (CLASS_NUM *
68                                     SAMPLE_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
69
70     # resize the images from 15 X 5 to 5 X 5 to get square images
71     # interpolate down samples the input to the given size
72     sample_features = F.interpolate(sample_features, size = 5)
73     sample_features = sample_features.view(CLASS_NUM, SAMPLE_NUM_PER_CLASS,
74                                           FEATURE_DIM, 5, 5) # CLASS_NUM X SAMPLE_NUM_PER_CLASS X FEATURE_DIM
75                                           X 5 X 5
76     sample_features = torch.sum(sample_features,1).squeeze(1) # CLASS_NUM
77                                           X FEATURE_DIM X 5 X 5
78
79     batch_features = feature_encoder(Variable(batches)) # (CLASS_NUM *
80                                     BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
81     batch_features = F.interpolate(batch_features, size = 5)
82
83     # calculate relations
84     # each batch sample link to every samples to calculate relations
85     sample_features_ext = sample_features.unsqueeze(0).repeat(CLASS_NUM *
86                                                             BATCH_NUM_PER_CLASS, 1, 1, 1) # (CLASS_NUM * BATCH_NUM_PER_CLASS)
87                                                             X 5 X FEATURE_DIM X 5 X 5
88     batch_features_ext = batch_features.unsqueeze(0).repeat(CLASS_NUM, 1, 1,
89                                                            1, 1) # 5 X (CLASS_NUM * BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X
90                                                            5
91     batch_features_ext = torch.transpose(batch_features_ext, 0, 1) # (
92                                     CLASS_NUM * BATCH_NUM_PER_CLASS) X 5 X FEATURE_DIM X 5 X 5
93
94     relation_pairs = torch.cat((sample_features_ext, batch_features_ext), 2).
95                             view(-1, FEATURE_DIM*2, 5, 5) # (CLASS_NUM * BATCH_NUM_PER_CLASS *
96                             5) X (FEATURE_DIM * 2) X 5 X 5
97     relations = relation_network(relation_pairs).view(-1, CLASS_NUM) # (
98                                     CLASS_NUM * BATCH_NUM_PER_CLASS) X CLASS_NUM
99
100     mse = nn.MSELoss()
101
102     one_hot_labels = Variable(torch.zeros(BATCH_NUM_PER_CLASS*CLASS_NUM,
103                                           CLASS_NUM))
104
105     for i in range(CLASS_NUM):
106         for j in range(BATCH_NUM_PER_CLASS):

```

```

89         one_hot_labels[BATCH_NUM_PER_CLASS*i+j,i] = 1
90
91     if torch.cuda.is_available():
92         mse = mse.to(device='cuda')
93         one_hot_labels = one_hot_labels.to(device='cuda')
94
95     loss = mse(relations, one_hot_labels)
96
97     # training
98
99     feature_encoder.zero_grad()
100    relation_network.zero_grad()
101
102    loss.backward()
103
104    torch.nn.utils.clip_grad_norm_(feature_encoder.parameters(), 0.5)
105    torch.nn.utils.clip_grad_norm_(relation_network.parameters(), 0.5)
106
107    feature_encoder_optim.step()
108    relation_network_optim.step()
109
110    train_loss.append(loss.item())
111
112    if (episode+1)%5000 == 0:
113        total_rewards = 0
114        #for i in tqdm(range(TEST_EPISODE), desc = "validation episode",
115            position = 1, leave = False):
116            for validation_episode in range(TEST_EPISODE):
117                # sample datas
118                batches, samples = batch_sample(validation_readers, CLASS_NUM,
119                    SAMPLE_NUM_PER_CLASS, BATCH_NUM_PER_CLASS) # samples: C X K X
120                    128 X 51, batches: C X Q X 128 X 51
121
122                samples = samples.view(CLASS_NUM * SAMPLE_NUM_PER_CLASS, 1, *samples
123                    .size()[2:]) # (C X K) X 1 X 51 X 51
124                batches = batches.view(CLASS_NUM * BATCH_NUM_PER_CLASS, 1, *batches.
125                    size()[2:]) # (C X Q) X 1 X 51 X 51
126
127                if torch.cuda.is_available():
128                    samples = samples.to(device)
129                    batches = batches.to(device)
130
131                # calculate features
132                sample_features = feature_encoder(Variable(samples)) # (CLASS_NUM *
133                    SAMPLE_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
134
135                # resize the images from 15 X 5 to 5 X 5 to get square images
136                # interpolate down samples the input to the given size
137                sample_features = F.interpolate(sample_features, size = 5)
138                sample_features = sample_features.view(CLASS_NUM,
139                    SAMPLE_NUM_PER_CLASS, FEATURE_DIM, 5, 5) # CLASS_NUM X
140                    SAMPLE_NUM_PER_CLASS X FEATURE_DIM X 5 X 5
141                sample_features = torch.sum(sample_features,1).squeeze(1) #
142                    CLASS_NUM X FEATURE_DIM X 5 X 5
143
144                batch_features = feature_encoder(Variable(batches)) # (CLASS_NUM *
145                    BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
146                batch_features = F.interpolate(batch_features, size = 5)
147
148                # calculate relations
149                # each batch sample link to every samples to calculate relations
150                sample_features_ext = sample_features.unsqueeze(0).repeat(CLASS_NUM

```



```

141         * BATCH_NUM_PER_CLASS, 1, 1, 1, 1) # (CLASS_NUM *
        BATCH_NUM_PER_CLASS) X 5 X FEATURE_DIM X 5 X 5
142     batch_features_ext = batch_features.unsqueeze(0).repeat(CLASS_NUM,
        1, 1, 1, 1) # 5 X (CLASS_NUM * BATCH_NUM_PER_CLASS) X
        FEATURE_DIM X 5 X 5
143     batch_features_ext = torch.transpose(batch_features_ext, 0, 1) # (
        CLASS_NUM * BATCH_NUM_PER_CLASS) X 5 X FEATURE_DIM X 5 X 5
144
145     relation_pairs = torch.cat((sample_features_ext, batch_features_ext),
        2).view(-1, FEATURE_DIM*2, 5, 5) # (CLASS_NUM *
        BATCH_NUM_PER_CLASS * 5) X (FEATURE_DIM * 2) X 5 X 5
146     relations = relation_network(relation_pairs).view(-1, CLASS_NUM) # (
        CLASS_NUM * BATCH_NUM_PER_CLASS) X CLASS_NUM
147
148     _, predict_labels = torch.max(relations.data, 1)
149
150     test_labels = torch.arange(CLASS_NUM).expand(BATCH_NUM_PER_CLASS,
        CLASS_NUM).transpose(1, 0).reshape(-1)
151
152     rewards = [1 if predict_labels[j]==test_labels[j] else 0 for j in
        range(CLASS_NUM*BATCH_NUM_PER_CLASS)]
153
154     total_rewards += np.sum(rewards)
155
156     valid_accuracy = total_rewards/1.0/CLASS_NUM/BATCH_NUM_PER_CLASS/
        TEST_EPISODE
157
158     print("\nTest accuracy: {}".format(valid_accuracy))
159
160     if valid_accuracy > last_accuracy:
161         torch.save({
162             'epoch': episode+1,
163             'feature_encoder_state_dict': feature_encoder.state_dict(),
164             'relation_network_state_dict': relation_network.
                state_dict(),
165             'feature_encoder_optim_state_dict':
                feature_encoder_optim.state_dict(),
166             'relation_network_optim_state_dict':
                relation_network_optim.state_dict(),
167             'train_loss': train_loss,
168             'avg_loss_tr': np.mean(train_loss),
169             'valid_acc': valid_accuracy,
170             'avg_acc_val': np.mean(valid_accuracy),
171             }, "/Modelli-Relation-Network/relation_model_C{}_K{}.pt"
                .format(CLASS_NUM, SAMPLE_NUM_PER_CLASS))
172         last_accuracy = valid_accuracy
173
174     scipy.io.savemat('/Modelli-Relation-Network/relation_results_C{}_K
        {}'.mat'.format(CLASS_NUM, SAMPLE_NUM_PER_CLASS), {'train_loss':
        train_loss, 'valid_accuracy': valid_accuracy})
175
176     print('Average train loss: {}'.format(np.mean(train_loss)))

```

Codice 26: relation\_training.py

### 8.3 Validation

Come nel capitolo 7.3, ogni 5000 episodi di training ne vengono effettuati 1000 di validation. Allo stesso modo del validation della prototypical network, vengono utilizzati lettori di validation sconosciuti alla rete e, nel caso in cui la validation produca una accuracy migliore di quella precedente, il modello viene salvato. Per fare ciò viene utilizzato un array `reward` che contiene 1 se la label della  $j$ -esima predizione è uguale alla label della  $j$ -esima query, altrimenti 0. Successivamente, vengono sommati gli elementi di questo vettori e salvati nella variabile `total_rewards`. Per ogni episodio di validation, all'aumentare del numero di predizioni corrette, la variabile `total_rewards` tende a  $C \cdot Q$ .

L'accuracy equivale al numero di volte in cui la predizione è corretta, cioè `total_rewards`, diviso il totale delle  $C \cdot Q$  query, mediato nei 1000 episodi di validation, come mostrato nella formula (6).

$$\text{accuracy} = \frac{\text{total rewards}}{C \cdot Q \cdot 1000} \quad (6)$$

### 8.4 Test

Anche in questo il test ha come oggetto di analisi i singoli audio del dataset e non i lettori. Le fasi del test sono analoghe alla Prototypical network, ma cambia la funzione che calcola le predizioni `test_predictions`. Come in precedenza, il positive set, il negative set e il query set sono passati in ingresso alla prima rete che ne calcola gli embedding.

```
99 def main():
100     p = 5
101     n = 10
102
103     C = 2
104     K = 5
105
106     FEATURE_DIM = 64
107     RELATION_DIM = 8
108
109     feature_encoder = CNNEncoder()
110     relation_network = RelationNetwork(FEATURE_DIM, RELATION_DIM)
111
112     if torch.cuda.is_available():
113         checkpoint = torch.load("Models/Relation/relation_model_C{}_K{}.pt".
114                                 format(C, K), map_location=torch.device('cuda'))
115     else:
116         checkpoint = torch.load("Models/Relation/relation_model_C{}_K{}.pt".
117                                 format(C, K), map_location=torch.device('cpu'))
118     feature_encoder.load_state_dict(checkpoint['feature_encoder_state_dict'])
119     relation_network.load_state_dict(checkpoint['relation_network_state_dict'])
```

```

120     feature_encoder.eval()
121     relation_network.eval()
122
123     if torch.cuda.is_available():
124         feature_encoder.to(device='cuda')
125         relation_network.to(device='cuda')
126
127     auc_list = []
128     for audio in tqdm(os.listdir("Test_features/"), desc = "Test features"):
129         y_pred = []
130         y_true = []
131         # getting the number of target keywords in each audio
132         target_keywords_number = len([name for name in os.listdir(audio) if
133                                     os.path.isfile(os.path.join(audio, name))])
134
135         for i in range(target_keywords_number):
136             for j in range(1):
137                 negative_set, positive_set, query_set =
138                     get_negative_positive_query_set(p, n, i, audio)
139
140                 if torch.cuda.is_available():
141                     negative_set = negative_set.to(device='cuda')
142                     positive_set = positive_set.to(device='cuda')
143                     query_set = query_set.to(device='cuda')
144
145                 negative_set = negative_set.view(1 * n, 1, *negative_set.
146                                                  size()[1:])
147                 positive_set = positive_set.view(1 * p, 1, *positive_set.
148                                                  size()[1:])
149                 query_set = query_set.view(int(C * query_set.size()[0]/2),
150                                           1, *query_set.size()[1:]) # (C X Q) X 1 X 51 X 51
151
152                 y_pred_tmp, y_true_tmp = test_predictions(feature_encoder,
153                                                         relation_network, positive_set, negative_set, query_set,
154                                                         n, p)
155
156                 y_pred.extend(y_pred_tmp)
157                 y_true.extend(y_true_tmp)
158
159                 precision, recall, thresholds = sklearn.metrics.
160                     precision_recall_curve(y_true, y_pred)
161                 auc_tmp = sklearn.metrics.auc(recall, precision)
162                 auc_list.append(auc_tmp)
163
164     auc = np.mean(auc_list)
165     print("Area under precision recall curve: {}".format(auc))
166     auc_std_dev = np.std(auc_list)
167     print("Standard deviation area under precision recall curve: {}".format(
168         auc_std_dev))

```

Codice 27: relation\_test.py

Nella funzione `test_predictions` la dimensione del tensore in uscita dalla prima rete passa da  $(C \cdot Q) \times 64 \times 15 \times 5$  a  $(C \cdot Q) \times 64 \times 5 \times 5$ . Successivamente, il positive e negative set vengono concatenati in quanto svolgono il ruolo delle classi con cui confrontare la query. Tutte le query vengono quindi concatenate con i vettori rappresentanti delle due classi producendo un numero di concatenazioni pari al numero di query moltiplicato per 2 (ovvero

il doppio delle istanze rimanenti nella classe positiva dopo la creazione del positive set).

Questo tensore viene poi passato alla rete che calcola le predizioni su quale delle classi (positive set e negative set) sia più simile alla query. In questo modo la rete produce una coppia di valori che indicano la probabilità con cui una query è associata ai due set. Di questi viene presa solo l'elemento 0 che indica l'affinità con il positive set. Utilizzando poi un corrispettivo valore che contiene il valore atteso reale è possibile calcolare precision, recall e AUPRC.

```

1  def test_predictions(feature_encoder, relation_network, positive_set,
2      negative_set, query_set, n, p):
3      n_class = 2
4      n_query = query_set.size(0)
5
6      FEATURE_DIM = 64
7
8      pos_embeddings = feature_encoder(Variable(positive_set)) # (CLASS_NUM *
9          SAMPLE_NUM_PER_CLASS) X FEATURE_DIM X 15 X 5
10
11     # resize the images from 15 X 5 to 5 X 5 to get square images
12     # interpolate down samples the input to the given size
13     pos_embeddings = F.interpolate(pos_embeddings, size = 5)
14     pos_embeddings = pos_embeddings.view(1, p, FEATURE_DIM, 5, 5) #
15         CLASS_NUM X SAMPLE_NUM_PER_CLASS X FEATURE_DIM X 5 X 5
16     pos_embeddings = torch.sum(pos_embeddings, 1).squeeze(1) # CLASS_NUM X
17         FEATURE_DIM X 5 X 5
18
19     neg_embeddings = feature_encoder(Variable(negative_set)) # (CLASS_NUM *
20         SAMPLE_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
21
22     # resize the images from 15 X 5 to 5 X 5 to get square images
23     # interpolate down samples the input to the given size
24     neg_embeddings = F.interpolate(neg_embeddings, size = 5)
25     neg_embeddings = neg_embeddings.view(1, n, FEATURE_DIM, 5, 5) #
26         CLASS_NUM X SAMPLE_NUM_PER_CLASS X FEATURE_DIM X 5 X 5
27     neg_embeddings = torch.sum(neg_embeddings, 1).squeeze(1) # CLASS_NUM X
28         FEATURE_DIM X 5 X 5
29
30     batch_features = feature_encoder(Variable(query_set)) # (CLASS_NUM *
31         BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
32     batch_features = F.interpolate(batch_features, size = 5)
33
34     pos_neg_embeddings = torch.cat((pos_embeddings, neg_embeddings), 0)
35
36     # calculate relations
37     # each batch sample link to every samples to calculate relations
38     pos_neg_embeddings_ext = pos_neg_embeddings.unsqueeze(0).repeat(int(
39         n_class * n_query/2), 1, 1, 1, 1) # (CLASS_NUM * BATCH_NUM_PER_CLASS
40         ) X 5 X FEATURE_DIM X 5 X 5
41
42     batch_features_ext = batch_features.unsqueeze(0).repeat(n_class, 1, 1,
43         1, 1) # 5 X (CLASS_NUM * BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
44     batch_features_ext = torch.transpose(batch_features_ext, 0, 1) # (
45         CLASS_NUM * BATCH_NUM_PER_CLASS) X CLASS_NUM X FEATURE_DIM X 5 X 5
46
47     relation_pairs = torch.cat((pos_neg_embeddings_ext, batch_features_ext),
48         2).view(-1, FEATURE_DIM*2, 5, 5) # (CLASS_NUM *
49         BATCH_NUM_PER_CLASS * CLASS_NUM) X (FEATURE_DIM * 2) X 5 X 5

```

```

36     relations = relation_network(relation_pairs).view(-1,n_class) # (
        CLASS_NUM * BATCH_NUM_PER_CLASS) X CLASS_NUM
37
38     target_inds = torch.arange(n_class-1, -1, step = -1).view(n_class, 1).
        expand(n_class, int(n_query/2)).long()
39
40     return relations[:,0].view(-1).detach(), target_inds.reshape(1, -1).
        squeeze()

```

Codice 28: Funzione test\_predictions

## 9 Risultati

### 9.1 Area under precision-recall curve (AUPRC)

Nella fase di test, la rete elabora una predizione dell'output a partire da un ingresso noto che poi viene confrontata con il valore effettivo. Nel nostro caso è presente un positive set e un negative set, si tratta quindi di classificazione binaria. Il confronto tra predizione e label può produrre quattro risultati:

- Vero Negativo (TN): il valore reale è negativo e il valore predetto è negativo;
- Vero Positivo (TP): il valore reale è positivo e il valore predetto è positivo;
- Falso Negativo (FN): il valore reale è positivo e il valore predetto è negativo;
- Falso Positivo (FP): il valore reale è negativo e il valore predetto è positivo;

Questi valori vanno a comporre la confusion matrix. Nel nostro caso siamo interessati a Precision e Recall.

Il parametro Precision rappresenta quanti tra i casi predetti come positivi sono realmente positivi.

Il parametro Recall indica quanti tra i casi realmente positivi è stato predetto in modo corretto.

$$Precision = \frac{TP}{TP + FP} \quad (7)$$

$$Recall = \frac{TP}{TP + FN} \quad (8)$$

Consideriamo un vettore contenente le probabilità che una serie di query appartengano a una classe. Impostando una soglia di decisione e confrontando con un altro vettore contenente i valori desiderati, 1 se la query corrisponde alla classe positiva e 0 altrimenti, è possibile ottenere una coppia di valori di precision e recall.

A questo punto ordiniamo il vettore con valori decrescenti. Consideriamo come soglia il valore di probabilità presente al primo elemento e calcoliamo precision e recall. Poi, spostiamo la soglia al valore del secondo elemento e calcoliamo altri valori. Continuando in questo modo si può ottenere una curva considerando alle ascisse i valori di recall e come ordinate i valori di precision.

Calcolando l'area sottesa dalla curva si ricava il valore detto Area under precision-recall curve (AUPRC). Questo valore è molto utile quando i dati sono sbilanciati e, in una classificazione binaria, siamo più interessati al riconoscimento di una classe in particolare. La figura 9 mostra la curva precision-recall per la rete prototypical con  $C = 2$ ,  $K = 1$  e  $p = 5$ . Calcolando l'area sottesa dalla curva si può trovare il valore di AUPRC.

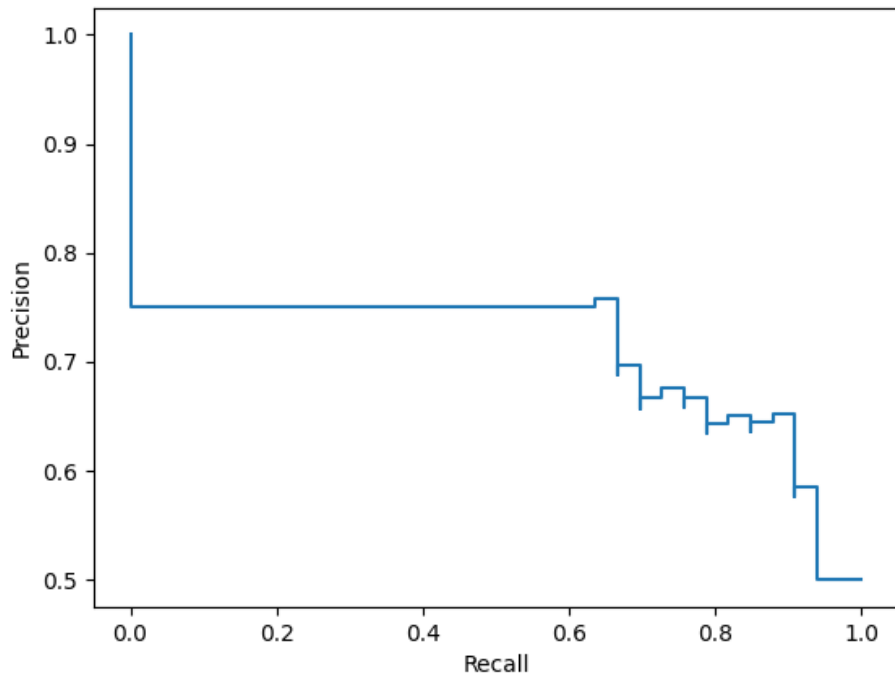


Figura 9: Curva precision-recall per il test della rete prototypical con  $C = 2$ ,  $K = 1$  e  $p = 5$ .

## 9.2 Confronto con l'AUPRC di riferimento

C, K	AUPRC	Deviazione standard	C, K	AUPRC	Deviazione standard
2, 1	0.731	0.069	2, 1	0.800	0.056
2, 5	0.721	0.074	2, 5	0.791	0.062
5, 1	0.709	0.089	5, 1	0.774	0.092
5, 5	0.688	0.062	5, 5	0.754	0.075
10, 1	0.728	0.067	10, 1	0.787	0.057
10, 5	0.727	0.078	10, 5	0.794	0.070
10, 10	0.725	0.059	10, 10	0.793	0.050

(a) Prototypical network  $p = 1$ ,  $n = 10$ . (b) Prototypical network  $p = 5$ ,  $n = 10$ .

Tabella 1: AUPRC e deviazione standard della prototypical network

C, K	AUPRC	Deviazione standard	C, K	AUPRC	Deviazione standard
2, 1	0.775	0.069	2, 1	0.596	0.042
2, 5	0.609	0.046	2, 5	0.813	0.056
5, 1	0.889	0.061	5, 1	0.698	0.050
5, 5	0.666	0.047	5, 5	0.945	0.037
10, 1	0.898	0.053	10, 1	0.691	0.052
10, 5	0.710	0.059	10, 5	0.945	0.036
10, 10	0.538	0.015	10, 10	0.845	0.061

(a) Relation network  $p = 1$ ,  $n = 10$ . (b) Relation network  $p = 5$ ,  $n = 10$ .

Tabella 2: AUPRC e deviazione standard della relation network

La figura 10 mostra i risultati ottenuti per i diversi modelli allenati con differenti  $C$  e  $K$ . I test sono stati fatti con  $p \in \{1, 5\}$  e con  $n = 10$  per le due reti proposte.

La figura 11 mostra i risultati ottenuti nell'articolo [6].



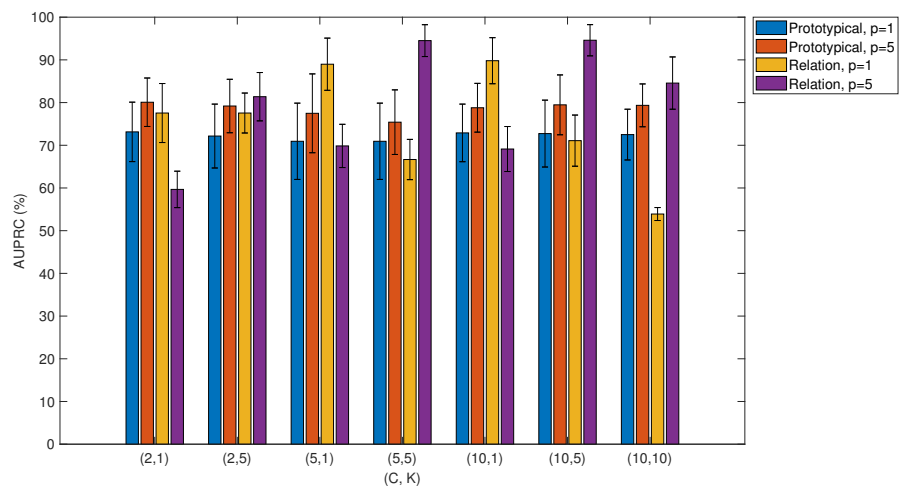


Figura 10: Risultati ottenuti.

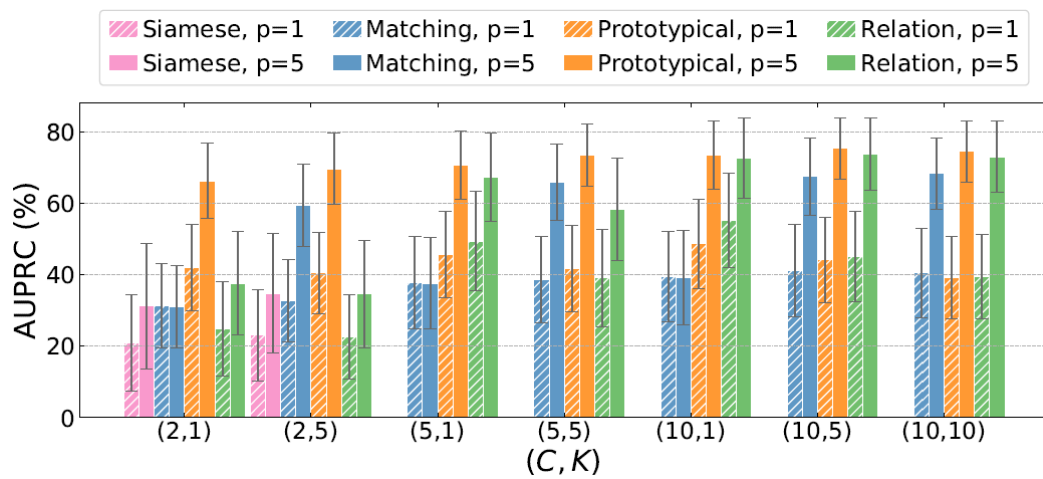


Figura 11: Risultati proposti in [6].

## 10 Conclusioni

## Riferimenti bibliografici

- [1] Gregory Koch, Richard Zemel e Ruslan Salakhutdinov. «Siamese Neural Networks for One-shot Image Recognition». In: 2015.
- [2] Arne Köhn, Florian Stegen e Timo Baumann. «Mining the Spoken Wikipedia for Speech Data and Beyond». Inglese. In: *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)* (23–28 mag. 2016). A cura di Nicoletta Calzolari (Conference Chair), Khalid Choukri, Thierry Declerck, Marko Grobelnik, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk e Stelios Piperidis. Portorož, Slovenia: European Language Resources Association (ELRA). ISBN: 978-2-9517408-9-1.
- [3] Jake Snell, Kevin Swersky e Richard S. Zemel. *Prototypical Networks for Few-shot Learning*. 2017. arXiv: 1703.05175 [cs.LG].
- [4] Flood Sung, Yongxin Yang, Li Zhang, Tao Xiang, Philip H. S. Torr e Timothy M. Hospedales. «Learning to Compare: Relation Network for Few-Shot Learning». In: *CoRR* abs/1711.06025 (2017). arXiv: 1711.06025. URL: <http://arxiv.org/abs/1711.06025>.
- [5] Oriol Vinyals, Charles Blundell, Timothy P. Lillicrap, Koray Kavukcuoglu e Daan Wierstra. «Matching Networks for One Shot Learning». In: *CoRR* abs/1606.04080 (2016). arXiv: 1606.04080. URL: <http://arxiv.org/abs/1606.04080>.
- [6] Yu Wang, Justin Salamon, Nicholas J. Bryan e Juan Pablo Bello. *Few-Shot Sound Event Detection*. 2020, pp. 81–85. DOI: 10.1109/ICASSP40776.2020.9054708.