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

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*" [4].

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 proposta, 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

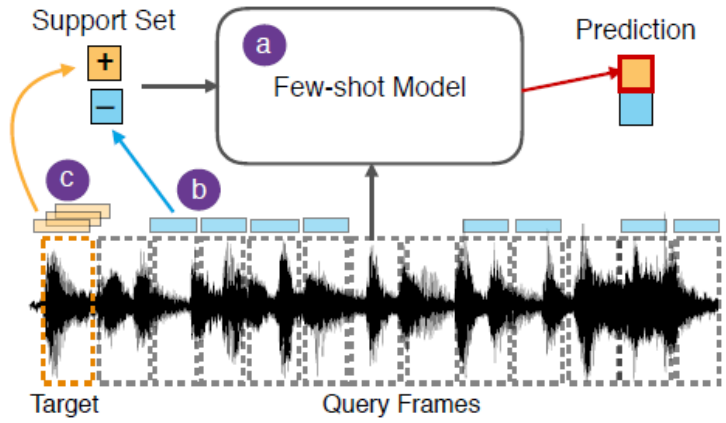


Figura 1: Metodo proposto per il few-shot sound event detection. (a) Applicazione del modello few-shot, (b) costruzione del set di esempi negativi, in blu, e (c) *data augmentation* per la generazione di più esempi positivi, in arancione. [4]

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.

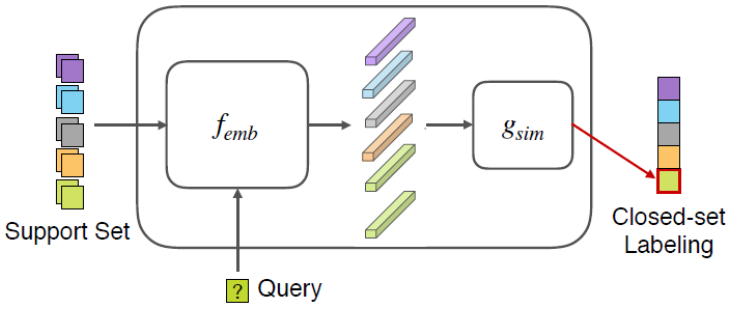


Figura 2: Modello few-shot learning nel caso 5-*way* 2-*shot*. [4]

Nell'articolo [4], 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, Matching Networks, Prototypical Networks [2] e Relation Networks [3].

La principale differenza tra questi metodi è la funzione g_{sim} o 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 Networks [2] e la Relation Networks [3], comparandone i risultati nel capitolo 9. Nel capitolo 2 vengono introdotte le reti neurali convoluzionali usate per implementare la funzionef_{emb}, in 3 vengono descritte le reti few-show 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) × (altezza dell'immagine) × (larghezza dell'immagine) × (canali dell'immagine). Dopo aver attraversato un layer convoluzionale, l'immagine viene astratta in una *feature map* di dimensione: (numero di input) × (altezza della feature map) × (larghezza della feature map) × (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 2×2) restituisce il valore massimo tra essi, che andrà a sostituire il blocco stesso. La figura 3 illustra un esempio di max pooling 2×2.

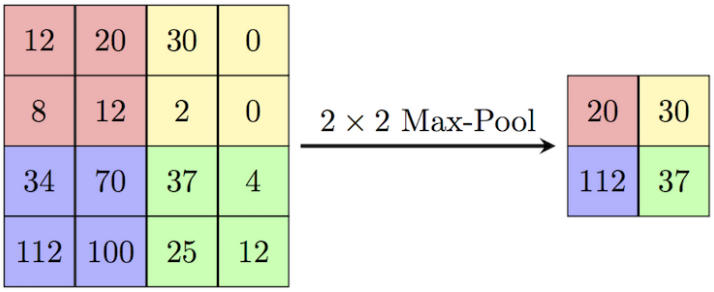


Figura 3: Esempio di max pooling 2×2.

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 performace 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 $\mathbf{c}_3$.

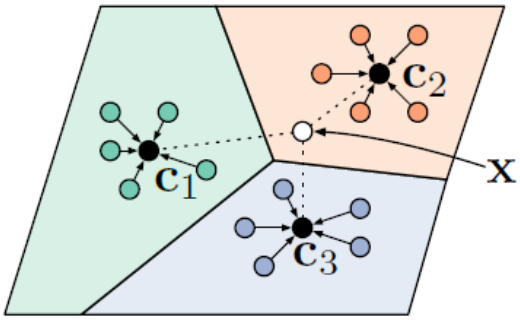


Figura 4: I prototipi few-shot $\mathbf{c}_k$ sono calcolati come la media degli embedding del support set per ogni classe. I punti degli embedding delle query sono classificati facendo il softmax sulle distanze del prototipo delle classi: $p_\phi(y = k|\mathbf{x}) \propto \exp\left(-d(f_\phi(\mathbf{x}), \mathbf{c}_k)\right)$. [2]

Nella classificazione few-shot per ogni episodio viene fornito un support set di N esempi etichettati $S = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_N, y_N)\}$ dove $\mathbf{x}_i \in \mathbb{R}^D$ rappresenta il vettore D-dimensionale della feature (nel nostro caso spettrogrammi di dimensione 128×51) e $y_i \in \{1, \ldots, K\}$ 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 $f_\phi : \mathbb{R}^D \to \mathbb{R}^M$ con parametri da allenare ϕ. 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.

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

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

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

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

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

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 Protypical 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 Protypical 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 Protypical 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 3×3, un layer di batch normalization, un layer ReLu che rappresenta la non linearità e un max-pooling 2×2.

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_φ (equivalente a quello nella Prototypical) e un modulo di *relation* g_ϕ, 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 $r_{i,j}$.

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

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.

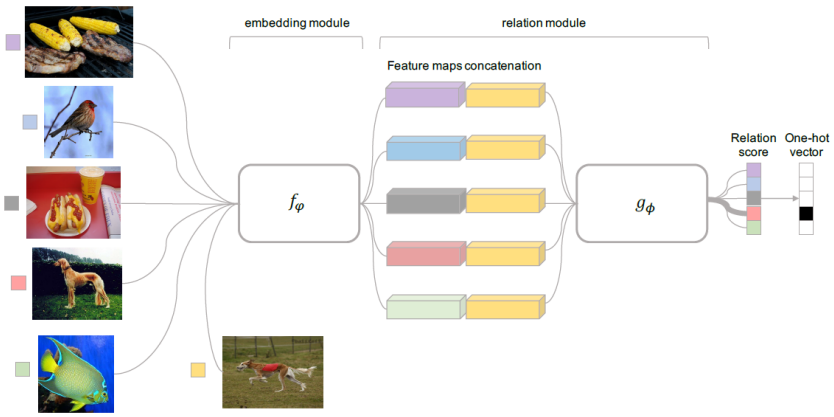
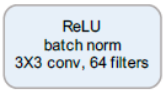


Figura 5: Architettura della Relation Network nel caso *5-way 1-shot* con un esempio di query. [3]

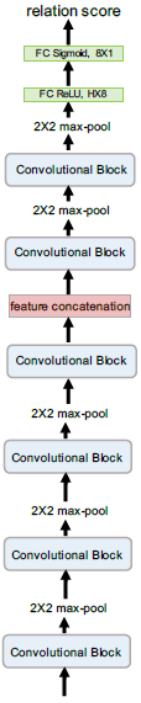
Come nella Prototypical Network, anche nella Relation Network viene usata una architettura con quattro blocchi convoluzionali per il modulo di embedding, come illustrato in figura 6.

Ogni blocco convoluzionale contiene una convoluzione con 64 filtri 3×3, una batch normalisation e un layer ReLU non lineare. I primi tre blocchi contengono anche un layer di max pooling 2×2, 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 3×3 con 64 filtri seguita da batch normalisation, non linearità ReLU e da un max pooling 2×2. 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.



(a) Blocco convoluzionale



(b) Relation Network per few-shot learning

Figura 6: Architettura della Relation Network per few-shot learning (b) composta dagli elementi inclusi nel blocco convoluzionale (a). [3]

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.[1]

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/a/
        aa/Limerence.ogg" group="audio1"/>
    <prop key="DC.source.audio.offset" value="0.0" group="audio1"/>
    <link key="DC.source.audio.page" value="https://en.wikipedia.org/w/index.php?title=
        File%3aLimerence.ogg" group="audio1"/>
```

```
    <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>
```

Codice 2: Annotazioni delle parole in un audio

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 [4] 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.

```python
import xml.etree.ElementTree as ET
from tqdm import tqdm
import os
import json
from json_manager import *

# Initialize list with empty dictionaries
readers = []

source_path = "./Dataset/English spoken wikipedia/english/"
filename = "aligned.swc"

# search for readers for each folder
for audio_path in os.scandir(source_path):
 # save only folder name from entire path
 folder =  os.path.basename(audio_path)
 if (os.path.exists(source_path + "/" + folder + "/" + filename)):
  # parse the xml file "aligned.swc"
  tree = ET.parse(source_path + "/" + folder + "/" + filename)
  # getroot returns the root element for this tree
  root = tree.getroot()
  #root.iter creates a tree iterator with the current element as the root.
  # The iterator iterates over this element and all elements below it, in
  # document (depth first) order.
  for property in root.iter(tag = 'prop'):
   # if the key "reader.name" exists
   if (property.attrib['key'] == 'reader.name'):
    # save the reader name taking the value of the attribute
    reader_name = property.attrib['value'].lower()
    # fix readers names that contain "user:"
    if ("user:" in reader_name):
    # fix readers names that contain "|""
     if ("|" in reader_name):
      # example reader_name = [[:en:user:alexkillby|alexkillby]] ->
      # reader_name = alexkillby
      reader_name = reader_name[reader_name.find("user:")+5:reader_name.find
          ("|")]
      # fix readers names that contain "|""
     elif ("]]" in reader_name):
      # example reader_name = [[user:popularoutcast]] ->
      # reader_name = popularoutcast
      reader_name = reader_name[reader_name.find("user:")+5:reader_name.find
          ("]]")]
    # if the reader is not yet on the list create a dict and append to
    # the readers list
    if not any(reader['reader_name'] == reader_name for reader in readers):
     dictionary = {'reader_name': reader_name, 'folder' : [folder]}
     readers.append(dictionary)
    else:
     # if the reader is already on the list add the folder name
     for reader in readers:
      if (reader['reader_name'] == reader_name):
       reader['folder'].append(folder)
# print the number of the readers
print("The readers are:", str(len(readers)))

# save a "readers_paths.json" with the name of the readers and the relative
# file audio folders
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. [4]

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.

```
import xml.etree.ElementTree as ET
from tqdm import tqdm
import os
import collections
from json_manager import *
import random

folder = []
source_path = "./Dataset/English spoken wikipedia/english/"
filename = "aligned.swc"
file_audio_name = "audio.ogg"

# iterate in each folder of the dataset
for folder in tqdm(os.scandir(source_path), desc = "Folder number"):
    # initialize the list of dict "target_words"
    target_words = []
    if (os.path.exists(folder.path + "/" + filename) \
        and os.path.exists(folder.path + "/" + file_audio_name)):
        # parse the xml file aligned.swc
        tree = ET.parse(folder.path + "/" + filename)
        # getroot returns the root element for this tree
        root = tree.getroot()
        # initialize the list "words"
        words = []
        # root.iter creates a tree iterator with the current element as the
        # root. The iterator iterates over this element and # all elements
        # below it, in document (depth first) order.
        for token_normalization in root.iter(tag = 'n'):
            # we take only the words with the timestamp, so only if there is
            # 'start' (or 'end') tag
            if 'start' in token_normalization.keys():
                # add every word with "start" key
                words.append(token_normalization.attrib['pronunciation'].
                    lower())
        # collections.Counter stores elements as dictionary keys, and their
        # counts are stored as dictionary values.
        unique_words = collections.Counter(words)
        # for each key (word) in "unique_words" append a new target_word if
        # the number of occurency is at least 10
        for key in unique_words.keys():
            # we only consider words that occur at least 10 times in the
            # recording. Note that unique_words[key] is the word frequency
            if (unique_words[key] >= 10):
                # add a new target word
                target_words.append({'word' : key, \
                                     'frequency' : unique_words[key], \
                                     'start' : [], \
                                     'end' : []})
        # for each "target_words" append the relative "start" and "end"
        # timestamp
        for token_normalization in root.iter(tag = 'n'):
            # we take only the words with the timestamp, so only if there is
            # 'start' (or 'end') tag
            if 'start' in token_normalization.keys():
                # iterate over the "target_words"
                for target_word in target_words:
                    # add start and end timestamp only to the relative
                    # "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.
               json")):
23             # "word_count.json" exists
24             json_file_exist = True
25             # read "word_count.json"
26             recording_words = read_json_file(source_path + "/" +
                   reader_folder + "/word_count.json")
27             # for each audio folder create a new list of dict
28             folder_per_word = []
29             # for each word in the audio save the folder, start and end
30             # timestamps
31             for word in recording_words:
32                  # create a dict with 'folder', 'start' and 'end' keys
33                 folder_per_word = {'folder' : reader_folder, \
34                                    'start' : word['start'], \
35                                    'end' : word['end']}
36                 # if the word is not yet in the list add the word
37                 if not any (word['word'] == word_per_reader['word'] for
                       word_per_reader in words_per_reader):
38                    words_per_reader.append({ 'word' : word['word'], \
39                                              'folders' : [folder_per_word]})
                                                   \
40                 # otherwise add 'start' and 'end' if the "reader_folder" is
```

```
                # in word_per_reader['folders'] or add the "folder_per_word"
                else:
                    for word_per_reader in words_per_reader:
                        if word['word'] == word_per_reader['word']:
                            if reader_folder in word_per_reader['folders']:
                                for folder in word_per_reader['folders']:
                                    if reader_folder == folder['folder']:
                                        folder['start'] += word['start']
                                        folder['end'] += word['end']
                            else:
                                word_per_reader['folders'].append(
                                    folder_per_word)
    # if "word_count.json" exists add the new reader to "training_readers"
    # list
    if (json_file_exist):
        new_readers_words['words'] = words_per_reader
        readers_words.append(new_readers_words)
# save a "readers_words.json" with the name of the readers and the relative
# words spoken
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,
              ...
```

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

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

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

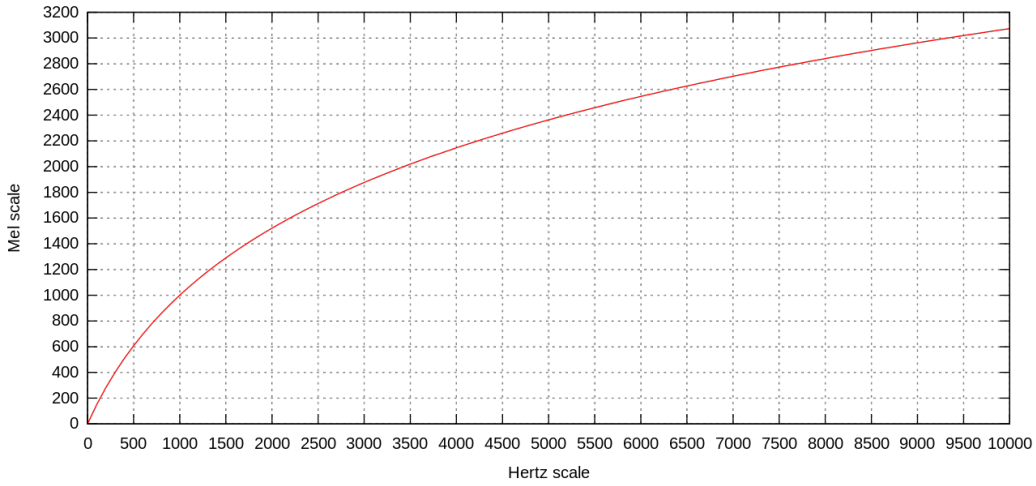


Figura 7: Grafico della scala mel in funzione della scala in Hertz.

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 128×51, poiché il numero di bande mel è 128 e la parola di mezzo secondo viene divisa in finestre 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 50.35 s e 50.78 s 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 [4] 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

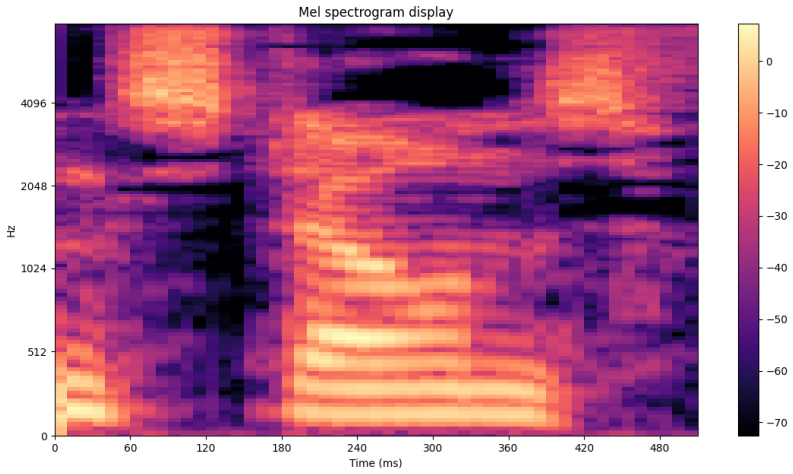


Figura 8: Spettrogramma di mezzo secondo centrato sulla parola “stones” pronunciata nell’audio “I can’t get no satisfaction” dopo 50.35 secondi.

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
```

```
# The readers are partitioned into training, validation, and test sets with a
# 138:15:30 ratio
number_of_training_readers = int(138/183*len(valid_readers))
number_of_test_readers = int(30/183*len(valid_readers))
number_of_validation_readers = int(15/183*len(valid_readers))

# The valid readers are partitioned into training, validation, and test
# readers
training_validation_readers, test_readers =
    create_training_validation_test_readers(valid_readers,
    number_of_training_readers, number_of_test_readers,
    number_of_validation_readers)
# create the folder for the training and validation features
training_validation_feature_folder_name = "Training_validation_features/"

if not (os.path.exists(training_validation_feature_folder_name)):
    try:
        os.mkdir(training_validation_feature_folder_name)
    except OSError as error:
        print(error)

max_class_number = 32
max_instances_number = 64

dataset_path = "./Dataset/English spoken wikipedia/english/"
audio_file_name = "audio.ogg"

training_validation_readers = read_json_file("training_validation_readers.
    json")

save_dataset(training_validation_readers,
    training_validation_feature_folder_name, dataset_path, audio_file_name,
    max_class_number, max_instances_number)

reader_paths = read_json_file('readers_paths.json')

# create the folder for the test features
test_feature_folder_name = "Test_features/"

if not (os.path.exists(test_feature_folder_name)):
    try:
        os.mkdir(test_feature_folder_name)
    except OSError as error:
        print(error)

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.

```
def find_valid_readers(min_classes = 2, min_instances_per_class = 26):
    """
    find_valid_readers returns a list of dict with 'word', 'start', 'end'
    and 'folders' keys for each reader name only for the readers with at
    least min_classes words and min_instances_per_class instances per word

    Parameters:
    min_classes (int) (default 2): minimum number of classes per each reader
    min_instances_per_class (int) (default 26): minimum number of instances
    per class

    Returns:
    valid_readers (list of dict): it contains only the readers with at least
    min_classes words and min_instances_per_class instances per word
    """
    # read the json with the words of each reader
    readers = read_json_file("readers_words.json")

    valid_readers = []

    # for each reader find if it's a valid reader only if he reads C words
    # for K + Q instances
    for reader in readers:
        valid_words = []
        is_valid_reader = False
        number_of_valid_words = 0

        # for each word of a reader find if there are for K + Q instances
        for word in reader['words']:
            start = []
            end = []
            folders = []

            for item in word['folders']:
                start += item['start']
                end += item['end']
                folders += [item['folder']]*len(item['start'])

            new_word = {'word'   : word['word'], \
                        'start'  : start, \
                        'end'    : end, \
                        'folders': folders}

            # if there are at least 26 (default) instances append a new
            # valid word
            if (len(new_word['start']) >= min_instances_per_class):
                number_of_valid_words += 1
                valid_words.append(new_word)

        # if the number of valid words is at least min_classes, then the
        # reader is valid
        if (number_of_valid_words >= min_classes):
            is_valid_reader = True

        # if the reader is valid then add the reader to the valid readers
        # list
        if (is_valid_reader):
            valid_readers.append({'reader_name' : reader['reader_name'],\
                                  'words': valid_words})

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

```
    'start', 'end', 'folders' values are lists. The i-th element of 'start'
    and 'end' value  corresponds to the i-th element of 'folder' value.
    If the reader contains less words than max_class_number then random
    sample the words, otherwise random sample max_class_number words.

    Parameters:
    reader (string): one reader from the json file of readers
    max_class_number (int): maximum class size
    max_instances_number (int): maximum instance size

    Returns:
    classes (list): a list of dict that has 'word', 'start',
    'end', 'folders' keys.
    """
    classes = []

    # taking at most C words from the reader
    # If the reader contains more words than C, random sample C words
    if (len(reader['words']) >= max_class_number):
        reader_words = random.sample(reader['words'], max_class_number)
    # If the reader contains less words than C, random sample the words
    else:
        reader_words = random.sample(reader['words'], len(reader['words']))

    for word in reader_words:
        # numpy.arange returns evenly spaced values within a given interval.
        # create an array of index to get the start, end and folder of the
        # same index
        index_array = list(numpy.arange(len(word['start'])))

        # taking at most max_instances_number instances from each word
        # If the word contains more instances than max_instances_number,
        # random sample max_instances_number instances
        if (len(index_array) >= max_instances_number):
            index_array = random.sample(index_array, max_instances_number)
        # If the word contains less instances than max_instances_number,
        # random sample the index_array
        else:
            index_array = random.sample(index_array, len(index_array))

        instance_start = []
        instance_end = []
        instance_folder = []

        # sample K instances from every C word class
        for index in index_array:
            # get the start, end and folder of the same index
            instance_start.append(word['start'][index])
            instance_end.append(word['end'][index])
            instance_folder.append(word['folders'][index])

        # append the new word of K + Q instances
        classes.append({'word'    : word['word'], \
                        'start'   : instance_start,\
                        'end'     : instance_end, \
                        'folders' : instance_folder})

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

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

```
def save_test_dataset(test_readers, reader_paths, test_feature_folder_name,
    dataset_path, audio_file_name):
    """
    save_test_dataset saves a pytorch tensor for each word of a test reader
        in a folder named
    as the audio of the test reader.

    Parameters:
    test_readers (list of dict): the list of the test readers saved in
        test_readers.json
    test_feature_folder_name (string): name of the folder in which save the
        features
    dataset_path (string): path of the Spoken Wikipedia Corpora dataset
    audio_file_name (string): name of the ".ogg" audio file

    Returns:
    """
    readers_list = []
    audio_folders = []

    for reader in test_readers:
        readers_list.append(reader['reader_name'])

    for reader_name in tqdm(readers_list, position = 0, desc = "test readers
        "):
        for item in reader_paths:
            if reader_name == item['reader_name']:
                audio_folders += item['folder']

    for audio_folder in tqdm(audio_folders, position = 0, desc = "audio
        folders"):
        if not (os.path.exists(test_feature_folder_name + audio_folder)):
            try:
                os.mkdir(test_feature_folder_name + audio_folder)
                path = dataset_path + audio_folder + "/target_words.json"
                words = read_json_file(path)
                for word in words:
                    spectrograms = np.empty([0, 128, 51])
                    if (len(word['start']) > 16):
                        indices = random.sample(list(numpy.arange(len(word['
                            start']))), 16)
                    else:
                        indices = random.sample(list(numpy.arange(len(word['
                            start']))), len(word['start']))
```

```
                    for i in indices:
                        start_in_sec = word['start'][i]/1000 # conversion
                            from milliseconds to seconds
                        end_in_sec = word['end'][i]/1000     # conversion
                            from milliseconds to seconds
                        # calculation of the center time of the word
                        word_center_time = (start_in_sec + end_in_sec)/2
                        # path of the audio file
                        audio_file_path = dataset_path + audio_folder +
                            audio_file_name
                        if (os.path.exists(audio_file_path)):
                            # compute the 128 bit log mel-spectrogram
                            item_spectrogram = compute_melspectrogram(
                                audio_file_path, word_center_time)
                            # construction of a spectrogram tensor
                            spectrograms = np.concatenate((spectrograms, [
                                item_spectrogram]), axis = 0)
                    # save the spectrograms tensor only if the first
                        dimension is higher than K + Q,
                    # that is when the word has at least K + Q instances
                    #if (spectrograms.shape[0] >= K + Q):
                        # conversion from numpy array to torch.FloatTensor
                    torch_tensor = torch.FloatTensor(spectrograms)
                    # save the torch tensor
                    torch.save(torch_tensor, test_feature_folder_name +
                        audio_folder + "/" + word['word'] + ".pt")

            except OSError as error:
                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 3×3 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 [4], 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):
```

```
        super(Protonet,self).__init__()
        self.encoder = nn.Sequential(
            conv_block(1,64),
            conv_block(64,64),
            conv_block(64,64),
            conv_block(64,64)
        )
    def forward(self,x):
        (num_samples,mel_bins, seq_len) = x.shape
        # tensor 3D to tensor 4D with 3rd dimension = 1
        x = x.view(-1,1,mel_bins,seq_len)
        x = self.encoder(x)
        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 [4]; 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.

```
from tqdm import trange, tqdm
import torch
import numpy as np
import scipy
from scipy import io


if torch.cuda.is_available():
  device = torch.device("cuda")
  print("Device: {}".format(device))
  print("Device name: {}".format(torch.cuda.get_device_properties(device).
      name))
else:
  device = torch.device("cpu")

C = 10 # classes
K = 5 # instances per class
Q = 16 # query set size

model = Protonet()
if torch.cuda.is_available():
  model.to(device='cuda')

print("Model parameters: {}".format(count_parameters(model)))

optim = torch.optim.Adam(model.parameters(), lr = 0.001)

training_readers, validation_readers = get_training_validation_readers("/
    Training_validation_features/", C)

print ("Training...")

last_accuracy = 0.0
```

```
train_loss = []
train_acc = []


# To construct a C-way K-shot training episode, we randomly sample a reader
# from the training set, sample C word classes from the reader, and sample K
# instances per class as the support set.

for episode in trange(60000, desc = "episode", position = 0, leave = True):
    query, support = batch_sample(training_readers, C, K, Q)
    if torch.cuda.is_available():
      support = support.to(device='cuda')
      query = query.to(device='cuda')

    model.train()
    optim.zero_grad()
    loss_out, acc_val = loss(support, query, model)

    loss_out.backward()
    optim.step()

    train_loss.append(loss_out.item())
    train_acc.append(acc_val.item())

    if (episode+1)%5000 == 0:
      valid_loss = []
      valid_acc = []
      print ("\nValidation...")

      model.eval()

      for validation_episode in range(1000):
          query, support = batch_sample(validation_readers, C, K, Q)
          if torch.cuda.is_available():
            support = support.to(device='cuda')
            query = query.to(device='cuda')

          val_loss, acc_val = loss(support, query, model)
          optim.step()

          valid_loss.append(val_loss.item())
          valid_acc.append(acc_val.item())

      print("\nValidation accuracy: {}".format(np.mean(valid_acc)))

      if np.mean(valid_acc) > last_accuracy:

        torch.save({
              'epoch': episode+1,
              'model_state_dict': model.state_dict(),
              'optimizer_state_dict': optim.state_dict(),
              'train_loss': train_loss,
              'train_acc' : train_acc,
              'valid_loss': valid_loss,
              'valid_acc' : valid_acc,
              'avg_loss_tr' : np.mean(train_loss),
              'avg_acc_tr' : np.mean(train_acc),
              'avg_loss_val' : np.mean(valid_loss),
              'avg_acc_val' : np.mean(valid_acc),
              }, "/Modelli-Prototypical-Network/prototypical_model_C{}_K{}.
                  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 [4]. 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 $C \times K \times 128 \times 51$.

```
import os
import numpy
import random

def batch_sample(features, C, K, Q = 16):
    """
    batch_sample returns the support and query set.
    It reads each folder in feature_folder and  load the tensor with the
        spectrograms of the word.
    Then random sample the instances (spectrograms) of the word to get only
        K+Q spectrograms.
    The first K spectrograms compose the support set and the last Q ones
        compose the query set.

    Parameters:
    features (list): training/validation features paths
    C (int): class size
    K (int): support set size
    Q (int): query set size (default: 16)

    Returns:
    support (torch.FloatTensor): support set
    query (torch.FloatTensor): query set
    """
    # initialize support tensor of dimension 0 x K x 128 x 51
    support = torch.empty([0, K, 128, 51])
    # initialize query tensor of dimension 0 x Q x 128 x 51
    query = torch.empty([0, Q, 128, 51])
    # random sample a reader
    reader = random.sample(features, 1)[0]
    words = []
    # scan the torch tensor saved in each reader folder
    for word in os.scandir(reader):
        # create a list containing the path of the words
        words.append(word.path)
    # random sample C paths of the words of a reader
    words = random.sample(words, C)
    # randomize the instances of each word
    for word in words:
        # load the tensor containing the spectrograms of the instances of
        # one word
        spectrogram_buf = torch.load(word)
        # get the spectrogram tensor shape
        x_dim, y_dim, z_dim = spectrogram_buf.shape
        # get the number of instances
        instances_number = (spectrogram_buf.shape)[0]
        # random sample K + Q indices
        index = random.sample(list(torch.arange(instances_number)), K + Q)
        # initialize the spectrogram tensor
        spectrogram = torch.empty([0, 128, 51])
        for i in index:
            # concatenate spectrogram_buf with spectrogram to get a new
```

```
            # tensor of random sampled instances of the word
            spectrogram = torch.cat((spectrogram, (spectrogram_buf[i, :, :])
                .view(1, y_dim, z_dim)), axis = 0)
        # concatenate the first K spectrograms with the support set
        support =  torch.cat((support, (spectrogram[:K]).view(1, K, y_dim,
            z_dim)), axis = 0)
        # concatenate the last Q spectrograms with the query set
        query = torch.cat((query, (spectrogram[K:K+Q]).view(1, Q, y_dim,
            z_dim)), axis = 0)
    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, $0, 1, \ldots, C$, 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 ($i = 1, \ldots, C \cdot Q$) e quelli del j-esimo ($j = 1, \ldots, C$) 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 lontata 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.

```
import torch
```

```
import torch.nn.functional as F

def euclidean_dist(x, y):
    """
    euclidean_dist computes the euclidean distance from two arrays x and y

    Parameters:
    x (torch.FloatTensor): query array
    y (torch.FloatTensor): prototype array

    Returns:
    torch.pow(x - y, 2).sum(2) (double): the euclidean distance from x and y
    """
    # x: N x D
    # y: M x D
    n = x.size(0)
    m = y.size(0)
    d = x.size(1)
    assert d == y.size(1)

    x = x.unsqueeze(1).expand(n, m, d)
    y = y.unsqueeze(0).expand(n, m, d)

    return torch.pow(x - y, 2).sum(2)

def loss(xs, xq, model):
    """
    loss returns the loss and accuracy value. It calculates p_y, the loss
    softmax over distances to the prototypes in the embedding space. We need
    to minimize the negative log-probability of p_y to proceed the learning
    process.

    Parameters:
    xs (torch.FloatTensor): support set
    xq (torch.FloatTensor): query set
    model (torch.nn.Module): neural network model

    Returns:
    loss_val (double): loss value
    acc_val (double): accuracy value
    """
    # xs = Variable()
    n_class = xs.size(0)
    assert xq.size(0) == n_class
    n_support = xs.size(1)
    n_query = xq.size(1)

    target_inds = torch.arange(0, n_class).view(n_class, 1, 1).expand(
        n_class, n_query, 1).long()

    if torch.cuda.is_available():
        target_inds = target_inds.to(device='cuda')

    x = torch.cat([xs.view(n_class * n_support, *xs.size()[2:]),
                   xq.view(n_class * n_query, *xq.size()[2:])], 0)

    embeddings = model(x)

    embeddings_dim = embeddings.size(-1)

    prototypes = embeddings[:n_class*n_support].view(n_class, n_support,
        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 [4], 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.scandir("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_` 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 postiva 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 numeri 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
                = 0)
66
67      for i in range(query_set.shape[0]):
68          query_sample = random.sample(words, 1)
69          spectrograms = torch.load(query_sample[0])
70          index = np.arange(spectrograms.shape[0])
71          query_index = random.sample(list(index), 1)
72          query = spectrograms[query_index, :, :]
73          query_set = torch.cat((query_set, query.view(1, 128, 51)), axis = 0)
74
75      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 dimesione 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
            (1)
23      pos_neg_embeddings =  torch.cat((positive_embeddings,
            negative_embeddings), 0)
24      query_embeddings = embeddings[p+n:]
25      dists = euclidean_dist(query_embeddings, pos_neg_embeddings)
26
27      target_inds = torch.arange(n_class-1, -1, step = -1).view(n_class, 1).
            expand(n_class, int(n_query/2)).long()
28
29      p_y = F.softmax(-dists, dim = 1).view(n_class, int(n_query/2), -1)
30
31      return p_y[:,:,0].view(-1).detach(), target_inds.reshape(1, -1).squeeze
            ()
```

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 3×3 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 [3] 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 predirre 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.
```

```
    The first 3 blocks also contain a 2 X 2 max-pooling layer while the last
    two do not. We do so because we need the output feature maps for further
    convolutional layers in the relation module
    """
    def __init__(self):
        super(CNNEncoder, self).__init__()
        self.layer1 = nn.Sequential(
                        nn.Conv2d(1,64,kernel_size=3,padding=0),
                        nn.BatchNorm2d(64, momentum=1, affine=True),
                        nn.ReLU(),
                        nn.MaxPool2d(2))
        self.layer2 = nn.Sequential(
                        nn.Conv2d(64,64,kernel_size=3,padding=0),
                        nn.BatchNorm2d(64, momentum=1, affine=True),
                        nn.ReLU(),
                        nn.MaxPool2d(2))
        self.layer3 = nn.Sequential(
                        nn.Conv2d(64,64,kernel_size=3,padding=1),
                        nn.BatchNorm2d(64, momentum=1, affine=True),
                        nn.ReLU(),
                        nn.MaxPool2d(2))
        self.layer4 = nn.Sequential(
                        nn.Conv2d(64,64,kernel_size=3,padding=1),
                        nn.BatchNorm2d(64, momentum=1, affine=True),
                        nn.ReLU())
                        #nn.MaxPool2d(2))

    def forward(self,x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        #out = out.view(out.size(0),-1)
        return out # 64

class RelationNetwork(nn.Module):
    """
    The RelationNetwork is the relation module. It consists of two
    convolutional blocks and two fully-connected layers. Each of
    convolutional block is a 3 X 3 convolution with 64 filters followed
    by batch normalisation, ReLU non-linearity and 2 X 2 max-pooling.
    The two fully-connected layers are 8 and 1 dimensional, respectively.
    All fully-connected layers are ReLU except the output layer is Sigmoid
    in order to generate relation scores in a reasonable range for all
    versions of our network architecture.
    """
    def __init__(self,input_size,hidden_size):
        super(RelationNetwork, self).__init__()
        self.layer1 = nn.Sequential(
                        nn.Conv2d(128,64,kernel_size=3,padding=1),
                        nn.BatchNorm2d(64, momentum=1, affine=True),
                        nn.ReLU(),
                        nn.MaxPool2d(2))
        self.layer2 = nn.Sequential(
                        nn.Conv2d(64,64,kernel_size=3,padding=1),
                        nn.BatchNorm2d(64, momentum=1, affine=True),
                        nn.ReLU(),
                        nn.MaxPool2d(2))
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

    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 5×5 a 2×2 e infine a 1×1. 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 concetenata 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()`.

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torch.nn.functional as F
from tqdm import trange, tqdm
import numpy as np
import scipy
from scipy import io

# Hyper Parameters
FEATURE_DIM = 64
RELATION_DIM = 8
CLASS_NUM = 5
SAMPLE_NUM_PER_CLASS = 1
BATCH_NUM_PER_CLASS = 16
EPISODE = 60000
TEST_EPISODE = 1000
LEARNING_RATE = 0.001

if torch.cuda.is_available():
  device = torch.device("cuda")
  print("Device: {}".format(device))
  print("Device name: {}".format(torch.cuda.get_device_properties(device).
      name))
else:
  device = torch.device("cpu")

training_readers, validation_readers = get_training_validation_readers("/
    Training_validation_features/", CLASS_NUM)

# init neural network
feature_encoder = CNNEncoder()
relation_network = RelationNetwork(FEATURE_DIM, RELATION_DIM)

feature_encoder.apply(weights_init)
relation_network.apply(weights_init)

if torch.cuda.is_available():
    feature_encoder.to(device='cuda')
    relation_network.to(device='cuda')

feature_encoder_optim = torch.optim.Adam(feature_encoder.parameters(), lr =
    LEARNING_RATE)
relation_network_optim = torch.optim.Adam(relation_network.parameters(), lr
    = LEARNING_RATE)

print("Training...")

```

```
train_loss = []
validation_loss = []

last_accuracy = 0.0

for episode in tqdm(range(EPISODE), desc = "training episode", position=0):

    # sample datas
    batches, samples = batch_sample(training_readers, CLASS_NUM,
        SAMPLE_NUM_PER_CLASS, BATCH_NUM_PER_CLASS) # samples: C X K X 128 X
        51,  batches: C X Q X 128 X 51

    samples = samples.view(CLASS_NUM * SAMPLE_NUM_PER_CLASS, 1, *samples.
        size()[2:])  # (C X K) X 1 X 51 X 51
    batches = batches.view(CLASS_NUM * BATCH_NUM_PER_CLASS, 1, *batches.size
        ()[2:])   # (C X Q) X 1 X 51 X 51

    if torch.cuda.is_available():
        samples = samples.to(device='cuda')
        batches = batches.to(device='cuda')

    # calculate features
    sample_features = feature_encoder(Variable(samples)) # (CLASS_NUM *
        SAMPLE_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5

    # resize the images from 15 X 5 to 5 X 5 to get square images
    # interpolate down samples the input to the given size
    sample_features = F.interpolate(sample_features, size = 5)
    sample_features = sample_features.view(CLASS_NUM, SAMPLE_NUM_PER_CLASS,
        FEATURE_DIM, 5, 5) # CLASS_NUM X SAMPLE_NUM_PER_CLASS X FEATURE_DIM
        X 5 X 5
    sample_features = torch.sum(sample_features,1).squeeze(1)   # CLASS_NUM
        X FEATURE_DIM X 5 X 5

    batch_features = feature_encoder(Variable(batches)) # (CLASS_NUM *
        BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
    batch_features = F.interpolate(batch_features, size = 5)

    # calculate relations
    # each batch sample link to every samples to calculate relations
    sample_features_ext = sample_features.unsqueeze(0).repeat(CLASS_NUM *
        BATCH_NUM_PER_CLASS, 1, 1, 1, 1) # (CLASS_NUM * BATCH_NUM_PER_CLASS)
         X 5 X FEATURE_DIM X 5 X 5
    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
    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

    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
    relations = relation_network(relation_pairs).view(-1,CLASS_NUM) #  (
        CLASS_NUM * BATCH_NUM_PER_CLASS) X CLASS_NUM

    mse = nn.MSELoss()

    one_hot_labels = Variable(torch.zeros(BATCH_NUM_PER_CLASS*CLASS_NUM,
        CLASS_NUM))

    for i in range(CLASS_NUM):
        for j in range(BATCH_NUM_PER_CLASS):
```

```
            one_hot_labels[BATCH_NUM_PER_CLASS*i+j,i] = 1

    if torch.cuda.is_available():
        mse = mse.to(device='cuda')
        one_hot_labels = one_hot_labels.to(device='cuda')

    loss = mse(relations,one_hot_labels)

    # training

    feature_encoder.zero_grad()
    relation_network.zero_grad()

    loss.backward()

    torch.nn.utils.clip_grad_norm_(feature_encoder.parameters(), 0.5)
    torch.nn.utils.clip_grad_norm_(relation_network.parameters(), 0.5)

    feature_encoder_optim.step()
    relation_network_optim.step()

    train_loss.append(loss.item())

    if (episode+1)%5000 == 0:
      total_rewards = 0
      #for i in tqdm(range(TEST_EPISODE), desc = "validation episode",
          position = 1, leave = False):
      for validation_episode in range(TEST_EPISODE):
        # sample datas
        batches, samples = batch_sample(validation_readers, CLASS_NUM,
            SAMPLE_NUM_PER_CLASS, BATCH_NUM_PER_CLASS) # samples: C X K X
            128 X 51,  batches: C X Q X 128 X 51

        samples = samples.view(CLASS_NUM * SAMPLE_NUM_PER_CLASS, 1, *samples
            .size()[2:])  # (C X K) X 1 X 51 X 51
        batches = batches.view(CLASS_NUM * BATCH_NUM_PER_CLASS, 1, *batches.
            size()[2:])   # (C X Q) X 1 X 51 X 51

        if torch.cuda.is_available():
            samples = samples.to(device)
            batches = batches.to(device)

        # calculate features
        sample_features = feature_encoder(Variable(samples)) # (CLASS_NUM *
            SAMPLE_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5

        # resize the images from 15 X 5 to 5 X 5 to get square images
        # interpolate down samples the input to the given size
        sample_features = F.interpolate(sample_features, size = 5)
        sample_features = sample_features.view(CLASS_NUM,
            SAMPLE_NUM_PER_CLASS, FEATURE_DIM, 5, 5) # CLASS_NUM X
            SAMPLE_NUM_PER_CLASS X FEATURE_DIM X 5 X 5
        sample_features = torch.sum(sample_features,1).squeeze(1)   #
            CLASS_NUM X FEATURE_DIM X 5 X 5

        batch_features = feature_encoder(Variable(batches)) # (CLASS_NUM *
            BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
        batch_features = F.interpolate(batch_features, size = 5)

        # calculate relations
        # each batch sample link to every samples to calculate relations
        sample_features_ext = sample_features.unsqueeze(0).repeat(CLASS_NUM
```

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

$$\text{accuracy} = \frac{\text{total rewards}}{C \cdot Q \cdot 1000} \tag{6}$$

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".
                format(C, K), map_location=torch.device('cuda'))
114     else:
115         checkpoint = torch.load("Models/Relation/relation_model_C{}_K{}.pt".
                format(C, K), map_location=torch.device('cpu'))
116     feature_encoder.load_state_dict(checkpoint['feature_encoder_state_dict'
            ])
117     relation_network.load_state_dict(checkpoint['relation_network_state_dict
            '])
118
119
```

```
    feature_encoder.eval()
    relation_network.eval()

    if torch.cuda.is_available():
        feature_encoder.to(device='cuda')
        relation_network.to(device='cuda')

    auc_list = []
    for audio in tqdm(os.scandir("Test_features/"), desc = "Test features"):
        y_pred = []
        y_true = []
        # getting the number of target keywords in each audio
        target_keywords_number = len([name for name in os.listdir(audio) if
            os.path.isfile(os.path.join(audio, name))])

        for i in range (target_keywords_number):
            for j in range (1):
                negative_set, positive_set, query_set =
                    get_negative_positive_query_set(p, n, i, audio)

                if torch.cuda.is_available:
                  negative_set = negative_set.to(device='cuda')
                  positive_set = positive_set.to(device='cuda')
                  query_set = query_set.to(device='cuda')

                negative_set = negative_set.view(1 * n, 1, *negative_set.
                    size()[1:])
                positive_set = positive_set.view(1 * p, 1, *positive_set.
                    size()[1:])
                query_set = query_set.view(int(C * query_set.size()[0]/2),
                    1, *query_set.size()[1:])   # (C X Q) X 1 X 51 X 51

                y_pred_tmp, y_true_tmp = test_predictions(feature_encoder,
                    relation_network, positive_set, negative_set, query_set,
                     n, p)

                y_pred.extend(y_pred_tmp)
                y_true.extend(y_true_tmp)

        precision, recall, thresholds = sklearn.metrics.
            precision_recall_curve(y_true, y_pred)
        auc_tmp = sklearn.metrics.auc(recall, precision)
        auc_list.append(auc_tmp)

    auc = np.mean(auc_list)
    print("Area under precision recall curve: {}".format(auc))
    auc_std_dev = np.std(auc_list)
    print("Standard deviation area under precision recall curve: {}".format(
        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.

```
def test_predictions(feature_encoder,relation_network, positive_set,
    negative_set, query_set, n, p):
    n_class = 2
    n_query = query_set.size(0)

    FEATURE_DIM = 64

    pos_embeddings = feature_encoder(Variable(positive_set)) # (CLASS_NUM *
        SAMPLE_NUM_PER_CLASS) X FEATURE_DIM X 15 X 5

    # resize the images from 15 X 5 to 5 X 5 to get square images
    # interpolate down samples the input to the given size
    pos_embeddings = F.interpolate(pos_embeddings, size = 5)
    pos_embeddings = pos_embeddings.view(1, p, FEATURE_DIM, 5, 5) #
        CLASS_NUM X SAMPLE_NUM_PER_CLASS X FEATURE_DIM X 5 X 5
    pos_embeddings = torch.sum(pos_embeddings,1).squeeze(1)   # CLASS_NUM X
        FEATURE_DIM X 5 X 5

    neg_embeddings = feature_encoder(Variable(negative_set)) # (CLASS_NUM *
        SAMPLE_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5

    # resize the images from 15 X 5 to 5 X 5 to get square images
    # interpolate down samples the input to the given size
    neg_embeddings = F.interpolate(neg_embeddings, size = 5)
    neg_embeddings = neg_embeddings.view(1, n, FEATURE_DIM, 5, 5) #
        CLASS_NUM X SAMPLE_NUM_PER_CLASS X FEATURE_DIM X 5 X 5
    neg_embeddings = torch.sum(neg_embeddings,1).squeeze(1)   # CLASS_NUM X
        FEATURE_DIM X 5 X 5

    batch_features = feature_encoder(Variable(query_set)) # (CLASS_NUM *
        BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
    batch_features = F.interpolate(batch_features, size = 5)

    pos_neg_embeddings =  torch.cat((pos_embeddings, neg_embeddings), 0)

    # calculate relations
    # each batch sample link to every samples to calculate relations
    pos_neg_embeddings_ext = pos_neg_embeddings.unsqueeze(0).repeat(int(
        n_class * n_query/2), 1, 1, 1, 1) # (CLASS_NUM * BATCH_NUM_PER_CLASS
        ) X 5 X FEATURE_DIM X 5 X 5

    batch_features_ext = batch_features.unsqueeze(0).repeat(n_class, 1, 1,
        1, 1)  # 5 X (CLASS_NUM * BATCH_NUM_PER_CLASS) X FEATURE_DIM X 5 X 5
    batch_features_ext = torch.transpose(batch_features_ext, 0, 1)  #  (
        CLASS_NUM * BATCH_NUM_PER_CLASS) X CLASS_NUM X FEATURE_DIM X 5 X 5

    relation_pairs = torch.cat((pos_neg_embeddings_ext,batch_features_ext),
        2).view(-1, FEATURE_DIM*2, 5, 5)  #  (CLASS_NUM *
        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.

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

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.

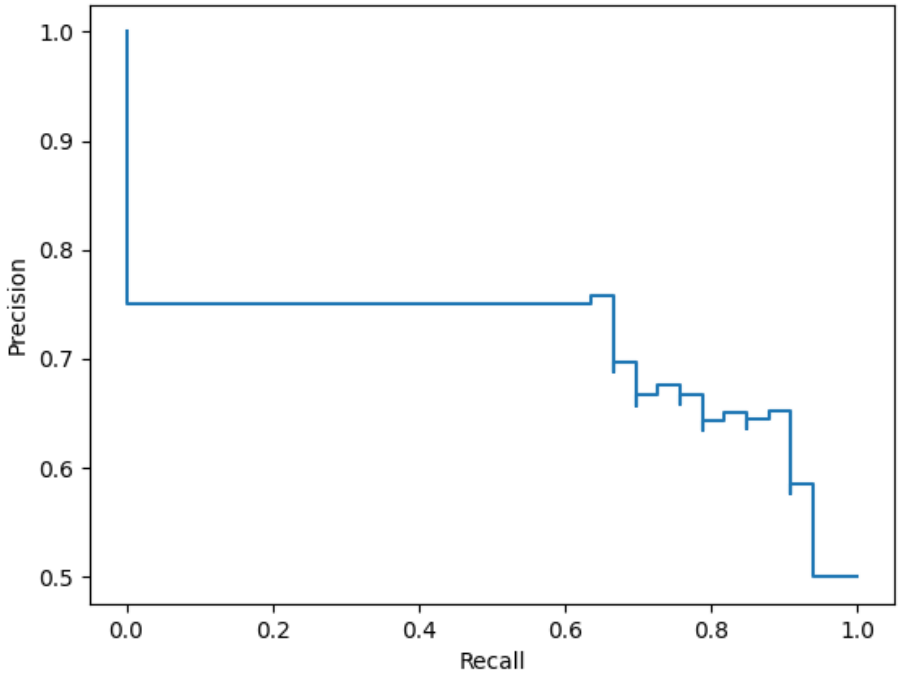


Figura 9: Curva precision-recall per il test della rete prototypical con $C = 2$, $K = 1$ e $p = 5$.

9.2 Confronto con l'AUPRC di riferimento

C, K	AUPRC	Deviazione standard
2, 1	0.731	0.069
2, 5	0.721	0.074
5, 1	0.709	0.089
5, 5	0.688	0.062
10, 1	0.728	0.067
10, 5	0.727	0.078
10, 10	0.725	0.059

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

C, K	AUPRC	Deviazione standard
2, 1	0.800	0.056
2, 5	0.791	0.062
5, 1	0.774	0.092
5, 5	0.754	0.075
10, 1	0.787	0.057
10, 5	0.794	0.070
10, 10	0.793	0.050

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

Tabella 1: AUPRC e deviazione standard della prototypical network

C, K	AUPRC	Deviazione standard
2, 1	0.775	0.069
2, 5	0.609	0.046
5, 1	0.889	0.061
5, 5	0.666	0.047
10, 1	0.898	0.053
10, 5	0.710	0.059
10, 10	0.538	0.015

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

C, K	AUPRC	Deviazione standard
2, 1	0.596	0.042
2, 5	0.813	0.056
5, 1	0.698	0.050
5, 5	0.945	0.037
10, 1	0.691	0.052
10, 5	0.945	0.036
10, 10	0.845	0.061

(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 [4].

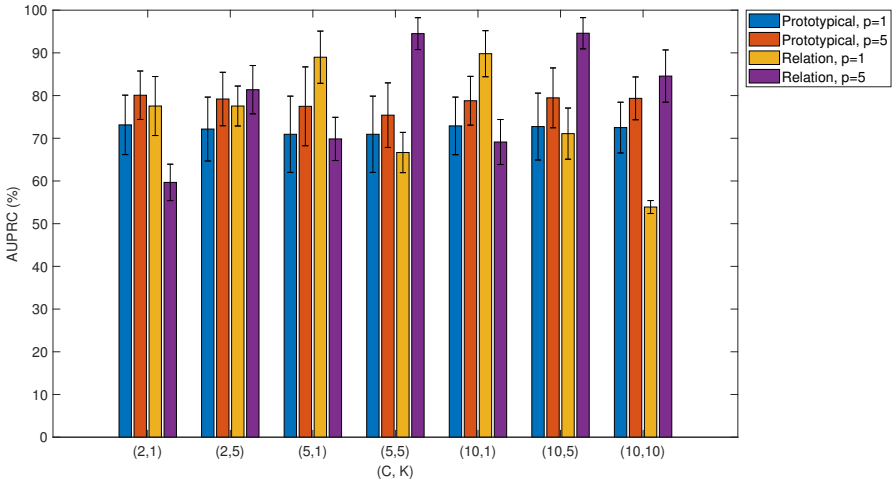


Figura 10: Risultati ottenuti.

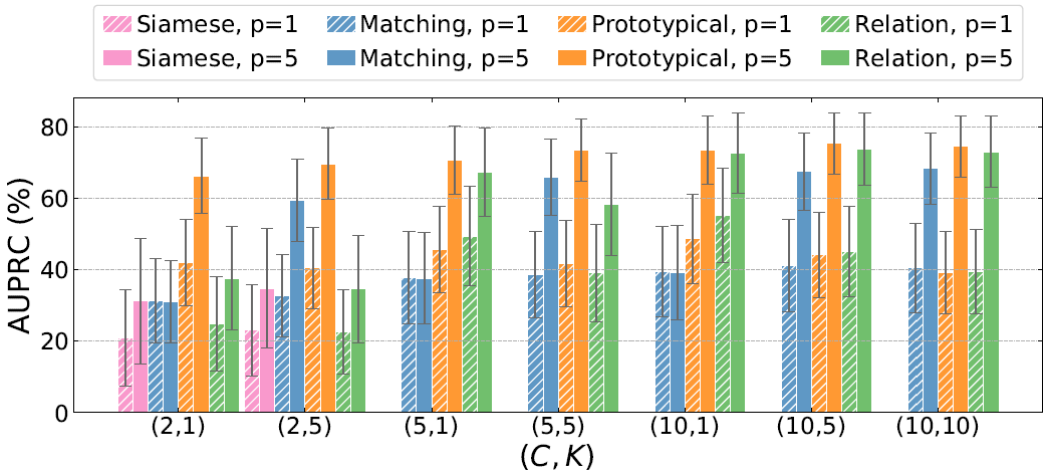


Figura 11: Risultati proposti in [4].

10 Conclusioni

Riferimenti bibliografici

[1] 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.

[2] Jake Snell, Kevin Swersky e Richard S. Zemel. *Prototypical Networks for Few-shot Learning*. 2017. arXiv: 1703.05175 [cs.LG].

[3] 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.

[4] 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.