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

16 luglio 2021

Indice

1 Introduzione

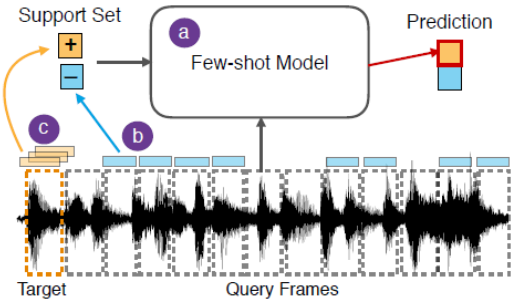


Figura 1: Metodo proposto per il few-shot sound event detection. (a) Applicazione del modello few-shot, (b) costruzione del set di esempi negativi, in blu, e (c) data augmentation per la generazioni di più esempi positivi, in arancione. [4]

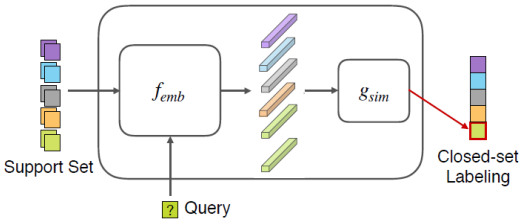


Figura 2: Modello few-shot learning nel caso 5-*way* 2-*shot*. [4]

2 Convolutional Neural Network (CNN)

Una rete neurale convoluzionale è una rete feedforward pensata per applicazioni che richiedono l'elaborazione di immagini o di grandi moli di dati. L'ingresso della rete è costituito da una o più matrici che attraversano delle fasi di convoluzione che ne riducono le dimensioni. I filtri di convoluzione sono detti kernel o di pooling. Infine, il risultato viene passato a una rete fully connected che ha il compito di classificazione.

La prima parte del filtro consiste nella convoluzione dell'immagine in ingresso con una serie di kernel i cui parametri vengono allenati. Il filtro viene traslato in entrambe le dimensioni della matrice producendo una serie di feature bidimensionali. 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 2x2) restituisce il valore massimo tra essi, che andrà a sostituire il blocco stesso.

3 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.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

4 Few-shot learning

(https://www.borealisai.com/en/blog/tutorial-2-few-shot-learning-and-meta-learning-i/)

4.1 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*). In pratica, il few-shot learning è utile quando è difficile trovare esempi di training (ad es. casi di una malattia rara) o quando il costo dell'etichettatura dei dati è elevato.

L'apprendimento few-shot viene solitamente studiato utilizzando la classificazione *N-way-K-shot*. L'obiettivo è quello di discriminare le N classi composte da K esempi ciascuna. Una tipica dimensione del problema potrebbe essere quella di discriminare tra $N = 10$ classi con solo $K = 5$ campioni ciascuno di training. Non possiamo allenare un classificatore usando metodi convenzionali; 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.

4.2 The meta learning framework

Nel framework di 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 in base a una serie di *training task* e valutiamo utilizzando una serie di test task; In altre parole, usiamo un insieme di problemi di classificazione per aiutare a risolvere altri insiemi non correlati.

Qui, ogni attività imita lo scenario few-shot, quindi per la classificazione N-way-K-shot, ogni attività include N classi con K esempi ciascuna. Questi sono noti come support set per il task e vengono utilizzati per apprendere come risolvere il task. Inoltre, esistono ulteriori esempi delle stesse classi, note come set di query, utilizzate per valutare le prestazioni del task corrente. Ogni task può essere completamente unico; potremmo non vedere mai le classi di un task in nessuno degli altri. L'idea è che il sistema veda ripetutamente istanze (task) durante l'addestramento che corrispondono alla struttura dell'attività finale di few-shot, ma che contengono classi diverse.

Ad ogni fase del meta-apprendimento, aggiorniamo i parametri del modello in base ad un training task selezionato casualmente. La funzione di loss è determinata dalle prestazioni di classificazione sul set di query del task, in base alla conoscenza acquisita dal relativo set di supporto. 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 in pochi colpi, utilizziamo una serie di attività di test. Ciascuno contiene solo classi invisibili che non erano in nessuna delle attività di formazione. Per ciascuno, misuriamo le prestazioni sul set di query in base alla conoscenza del loro set di supporto.

4.3 Approcci al meta-apprendimento

Gli approcci al meta-apprendimento sono diversi e non c'è consenso sull'approccio migliore. Tuttavia, esistono tre famiglie distinte, ognuna delle quali sfrutta un diverso tipo di conoscenza a priori:

- Conoscenze a priori sulla somiglianza: apprendiamo degli embedding nei training task che tendono a separare classi diverse anche quando non sono visibili.
- Conoscenze a priori sull'apprendimento: utilizziamo le conoscenze a priori per vincolare l'algoritmo di apprendimento a scegliere parametri che generalizzino bene da pochi esempi.

- Conoscenza a priori dei dati: sfruttiamo le conoscenze a priori sulla struttura e la variabilità dei dati e questo ci consente di apprendere modelli praticabili da pochi esempi.

5 Prototypical Network

L'approccio si basa sull'idea che esiste un embedding in cui i punti delle istanze di una classe si raggruppano attorno a una singola rappresentazione prototipo per ogni classe. 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.

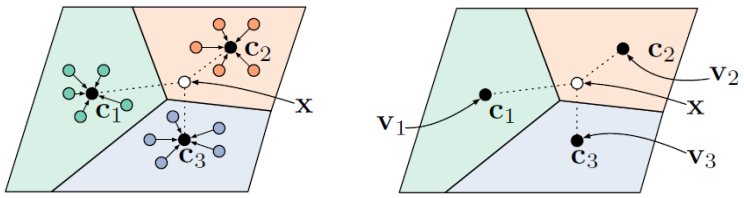
La rete Prototypical calcola una rappresentazione M-dimensionale $\mathbf{c}_k$, o *prototipo*, di ogni classe tramite una funzione di embedding $f_\phi : \mathbb{R}^D \to \mathbb{R}^M$ con parametri da allenare ϕ. La funzione di embedding è rappresentata da una rete convoluzionale. Ogni prototipo è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

$$\mathbf{c}_k = \frac{1}{|S_k|} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_\phi(\mathbf{x}_i) \tag{3}$$

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

$$p_\phi(y = k|\mathbf{x}) = \frac{\exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))}{\sum_k' \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k'))} \tag{4}$$

Il processo di training avviene minimizzando il negativo del logaritmo della probabilità $J(\phi) = -\log\left(p_\phi(y = k|\mathbf{x})\right)$ 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.



(a) Few-shot (b) Zero-shot

Figura 3: Reti Prototypical negli scenari few-shot e zero-shot. (a): i prototipi few-shot $\mathbf{c}_k$ sono calcolati come la media degli embedding del support set per ogni classe. (b): i prototipi zero-shot $\mathbf{c}_k$ sono prodotti facendo l'embedding dei metadati $\mathbf{v}_k$. In entrambi i casi 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]

6 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 4. 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.

$$r_{i,j} = g_\phi(\mathcal{C}(f_\varphi(x_i), f_\varphi(x_j))), \qquad i = 1, 2, \ldots, C \tag{5}$$

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.

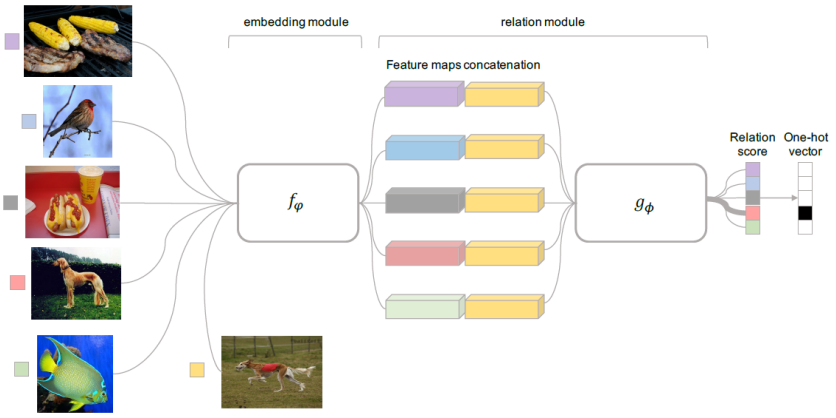
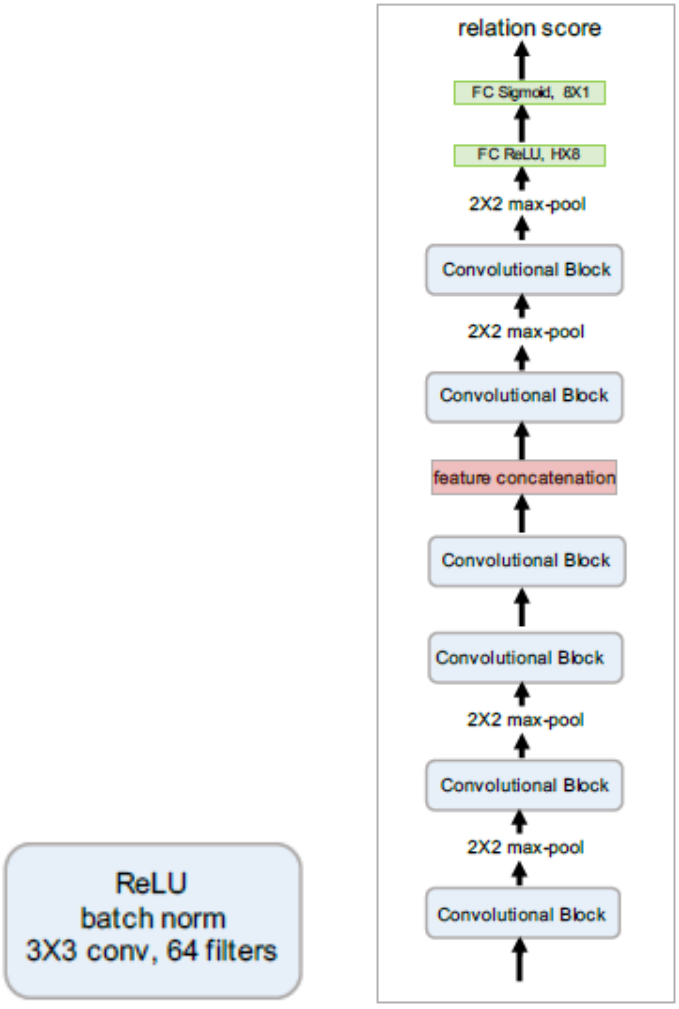


Figura 4: Architettura della Relation Network nel caso 5-*way* 1-*shot* con un esempio di query. [3]



(a) Blocco convoluzionale

(b) Relation Network per few-shot learning

Figura 5: Architettura della Relation Network per few-shot learning (b) composta dagli elementi inclusi nel blocco convoluzionale (a). [3]

7 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.

7.1 Annotazioni

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

7.2 Lettori

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.

```
import json

def write_json_file(filename, list_of_dict, indent = 0):
  f = open(filename, "w")
  f.write(json.dumps(list_of_dict, indent = indent))
  f.close

def read_json_file(filename):
  f = open(filename, "r")
  list_of_dict = json.load(f)
  f.close
  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 viene salvato nel file "aligned.swc" con nomi 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.

```
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

7.3 Parole

Il codice 6.....

```
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) \
        and os.path.isdir(folder.path)):
        # 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.
```

```
37          unique_words = collections.Counter(words)
38          # for each key (word) in "unique_words" append a new target_word if
39          # the number of occurency is at least 10
40          for key in unique_words.keys():
41              # we only consider words that occur at least 10 times in the
42              # recording. Note that unique_words[key] is the word frequency
43              if (unique_words[key] >= 10):
44                  # add a new target word
45                  target_words.append({'word' : key, \
46                                       'frequency' : unique_words[key], \
47                                       'start' : [], \
48                                       'end' : []})
49          # for each "target_words" append the relative "start" and "end"
50          # timestamp
51          for token_normalization in root.iter(tag = 'n'):
52              # we take only the words with the timestamp, so only if there is
                    'start' (or 'end') tag
53              if 'start' in token_normalization.keys():
54                  # iterate over the "target_words"
55                  for target_word in target_words:
56                      # add start and end timestamp only to the relative
57                      # "target_word"
58                      if target_word['word'] == token_normalization.attrib['
                            pronunciation'].lower():
59                          target_word['start'].append(int(token_normalization.
                                attrib['start']))
60                          target_word['end'].append(int(token_normalization.
                                attrib['end']))
61
62          write_json_file(folder.path+"/word_count.json", target_words, indent
                = 4)
63          # If there are more than 10 words that occur at least 10 times in
64          # the recording, we sort the words by their number of occurrences,
65          # divide the sorted list into 10 equally sized bins, and sample
66          # one keyword per bin.
67          if (len(target_words) >= 10):
68              target_words = random.sample(target_words, 10)
69          # save the "target_words.json"
70          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

Il file “readers_paths.json” viene successivamente usato 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 13 legge il nome di ogni lettore dal json precedentemente salvato, successivamente per ognuna delle cartelle del lettore si.....

```
from tqdm import tqdm
import os
from json_manager import *

source_path = "./Dataset/English spoken wikipedia/english/"
# read the json that contains the readers name and their audio folders
readers = read_json_file("readers_paths.json")
#initialize a list of dict
readers_words = []
# for each reader search the word spoken by the reader
for reader in tqdm(readers):
    # create a dict with 'reader_name' and 'words' keys
    new_readers_words = {'reader_name' : reader['reader_name'],\
                         'words' : [] }

    # for each reader create a new dict for words
    words_per_reader = []
    # flag to signal if "word_count.json" exists
    json_file_exist = False
    # search for each folder the file "word_count.json"
    for reader_folder in reader['folder']:
        if (os.path.exists(source_path + "/" + reader_folder + "/word_count.
            json")):
            # "word_count.json" exists
            json_file_exist = True
            # read "word_count.json"
            recording_words = read_json_file(source_path + "/" +
                reader_folder + "/word_count.json")
            # for each audio folder create a new list of dict
            folder_per_word = []
            # for each word in the audio save the folder, start and end
            # timestamps
            for word in recording_words:
                 # create a dict with 'folder', 'start' and 'end' keys
                folder_per_word = {'folder' : reader_folder, \
```

```
                                    'start' : word['start'], \
                                    'end' : word['end']}
                # if the word is not yet in the list add the word
                if not any (word['word'] == word_per_reader['word'] for
                    word_per_reader in words_per_reader):
                   words_per_reader.append({ 'word' : word['word'], \
                                             'folders' : [folder_per_word]})
                                                 \
                # 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

8 Protonet codice

```
import torch.nn as nn

def count_parameters(model):
return sum(p.numel() for p in model.parameters() if p.requires_grad)

def conv_block(in_channels,out_channels):

return nn.Sequential(
nn.Conv2d(in_channels,out_channels,3,padding=1),
nn.BatchNorm2d(out_channels),
nn.ReLU(),
nn.MaxPool2d(2)
)

class Protonet(nn.Module):
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
#print("x.shape:",x.shape)
# tensor 3D to tensor 4D with 3rd dimension = 1
x = x.view(-1,1,mel_bins,seq_len)
#print("x.view:",x.shape)
x = self.encoder(x)
#print("x.encoder:",x.shape)
return x.view(x.size(0),-1)
```

Codice 10: words_per_reader.py

La CNN al che funge da rete di embedding per la Prototypical Network è costituita da 4 strati convoluzionali. 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 3x3 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, in modo da avere embedding vettoriali.

8.1 Protonet training

```
def get_training_validation_readers(features_folder, C):
"""
get_training_validation_readers returns training and validation readers from the .
From the training and validation readers, it takes only the readers with at least C words
and split the list in training readers and validation readers.

Parameters:
features_folder (list of string): list of the path of the training and validation
C (int): number of classes

Returns:
training_readers (list of string): list of the training readers paths
validation_readers (list of string): list of the validation readers paths
"""
readers_path = []
train_val_readers = []
# scan each reader folder in the feature_folder
for entry in os.scandir(features_folder):
# create a list of reader names
readers_path.append(entry.path)
for reader_path in readers_path:
words = []
for word in os.scandir(reader_path):
# create a list containing the path of the words
words.append(word.path)
if (len(words) >= C):
train_val_readers.append(reader_path)
train_val_readers = random.sample(train_val_readers, len(train_val_readers))
training_readers = train_val_readers[:int(138/153*len(train_val_readers))]
validation_readers = train_val_readers[int(138/153*len(train_val_readers)):]
return training_readers, validation_readers
```

Codice 11: words_per_reader.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 [ARTICOLO RIFERIMENTO]: 138/153 e 15/153. 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 come spiegato sopra.

```
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 = 1 # 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("/
    content/drive/MyDrive/Few-Shot-Sound-Event-Detection/
    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),
}, "/content/drive/MyDrive/Few-Shot-Sound-Event-Detection/Modelli-
    Prototypical-Network/prototypical_model_C{}_K{}.pt".format(C, K))

last_accuracy = np.mean(valid_acc)

scipy.io.savemat('/content/drive/MyDrive/Few-Shot-Sound-Event-Detection/
    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 12: words_per_reader.py

Il primo for rappresenta l'inizio degli episodi. Il numero totale di iterazioni è 60000 come in [ARTICOLO]; per ogni episodio viene richiamata la funzione `batch_sample`. Essa 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. Viene inizializzato un vettore con gli indici di queste e ne vengono campionati K+Q in `index`. Poi, iterando, gli spettrogrammi il cui indice è presente in `index` vengono salvati e divisi in K istanze del support set e Q istanze del query set.

```
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 13: words_per_reader.py

Una volta ottenute, le feature del support set e query set vengono date in in-

gresso alla funzione `loss`. Queste ultime vengono messe in fila e concatenate 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: `prototypes = embeddings[:n_class*n_support].view(n_class, n_support, embeddings_dim).mean(1)`.

Il parametro `target_inds` viene costruito ripetendo l'indice di ogni classe per il numero di query e sarà utilizzato per poter ricordare a che classe appartiene ogni query.

Per ogni elemento del query set vengono calcolate le distanze rispetto ai prototipi di ogni classe tramite la funzione `euclidean_dist` che rappresenta la distanza euclidea. Per ogni query viene poi usata la funzione di attivazione `log_softmax` fra tutte le sue distanze. La funzione loss da minimizzare è costituita dalla media dei `log_softmax` della classe corrispondente alla query presi con segno negativo. Infatti, nel caso ideale in cui la distanza delle query della stessa classe dal prototipo è 0, la media dei softmax delle distanze dal prototipo assegnato sarà nulla. La accuracy invece è determinata dal numero dei casi in cui il prototipo a distanza minima da una query (o il massimo del softmax) corrisponde con il prototipo della classe associata alla query.

```
import torch
import torch.nn.functional as F
from torch.autograd import Variable

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()
    #target_inds = Variable(target_inds, requires_grad=False)

    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)

    queries = embeddings[n_class*n_support:]

    dists = euclidean_dist(queries, prototypes)

    log_p_y = F.log_softmax(-dists, dim=1).view(n_class, n_query, -1)

    loss_val = -log_p_y.gather(2, target_inds).squeeze().view(-1).mean()

    _, y_hat = log_p_y.max(2)
    acc_val = torch.eq(y_hat, target_inds.squeeze()).float().mean()

    return loss_val, acc_val
```

Codice 14: words_per_reader.py

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

8.2 Protonet 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 batch precedente il modello viene salvato e sovrascritto al precedente, altrimenti viene ignorato. In questo modo si aggira il problema dell'overfitting fungendo da pseudo early stopping.

8.3 Protonet test

Nella fase di test, seguendo l'articolo di riferimento, l'oggetto di interesse non sono i lettori come nel caso di training e validation. Vengono infatti presi in considerazione i singoli audio. 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 con spettrogramma (in base al contenuto di Test_features). 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 [ARTICOLO] 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. Per fare questo viene usata la funzione `get_negative_positive_query_set`. Tramite un indice che indica le istanze della parola "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.

A questo punto, i tre set e il modello sono parametri della funzione `test_loss`. 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 risulati viene selezionato il maggiore che rappresenta la predizione della rete: ovvero la classe a cui viene assegnata la query. Queste predizioni sono contenute nel vettore `y_pred_tmp`, mentre in `y_true_tmp` sono presenti le label reali di ogni query.

Prima di essere utilizzati per calcolare precision e recall, questi due vettori vengono sottoposti a una operazione di complemento a 2 che inverte 0 e 1

in quanto la funzione `precision_recall_curve` richiede che con la label 1 sia identificata la classe positiva. 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.

9 Relation codice

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 monodimensionali che possono essere confrontati. Anche in questo caso i layer sono 4 e sono costituiti da:

- `nn.Conv2d()`: Convoluzione bidimensionale con un kernel 3x3 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 [ARTICOLO RELATION] 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 dei quali viene eseguito un flattening che riduce l'uscita a un vettore. In seguito vengono applicati una ulteriore relu e una sigmoide.

9.1 Relation 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*Kx64x15x5 mentre quelli del query set sono C*Qx64x15x5.

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 5x5 a 2x2 a 1x1. In seguito al flattening le ultime due dimensioni scompaiono e l'ingresso alla relu ha la dimensione corretta: C*Q*Cx64. Le concatenazioni hanno dimensione C*Q*Cx64x5x5, infatti ogni query (CxQ) 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 vettore `relations` C*Q*Cx1 che viene poi suddiviso in una matrice C*QxC in cui in ogni riga sono presenti le probabilità che una data query appartenga alle varie classi. La matrice `one_hot_labels` ha la stessa forma 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 che viene utilizzata per aggiornare i parametri delle due reti con `loss.backward()`.

9.2 Relation validation

Ogni 5000 episodi di training ne vengono effettuati 1000 di validation. Come nella prototypical network vengono utilizzati lettori di validation sconosciuti alla rete e, nel caso in cui il batch produca una accuracy migliore di quello precedente, il modello viene salvato. Per fare ciò viene utilizzato un vettore `reward` che conta il numero di volte in cui il valore massimo tra le predizioni della classe da associare a una query corrisponde con quello reale. L'accuracy infatti equivale al numero di reward diviso il totale delle query, mediato nei 1000 episodi del batch di validation.

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.