PR3_PIT

March 10, 2022

1 PIT - Práctica 3: Detección de Actividad de Voz (VAD)

Alicia Lozano Díez y **Pablo Ramírez Hereza**

7 de marzo de 2022

1.1 Objetivo

El objetivo de esta práctica es proporcionar una introducción al procesamiento de señales temporales de voz, y desarrollar de un detector de actividad de voz basado en redes neuronales recurrentes, en particular, LSTM.

1.1.1 Materiales

- Guión (.ipynb) de la práctica - Moodle
- Ejemplos de datos y etiquetas - Moodle
- Listas de entrenamiento y validación - Moodle
- Scripts de descarga de datos - Moodle
- Datos y etiquetas de entrenamiento * - One Drive (https://dauam-my.sharepoint.com/:u:/g/personal/alicia_lozano_uam_es/EdCueYU7BpNAuo6BawH8hJAB5rclap745Bms
- Datos y etiquetas de validación * - One Drive (https://dauam-my.sharepoint.com/:u:/g/personal/alicia_lozano_uam_es/EWBjWyX774pLhJc2ahr4zk0BtLvWt7YGhdMI LcBNQ?e=sbgtjF)

CUIDADO: * Los datos proporcionados son de uso exclusivo para esta práctica. No tiene permiso para copiar, distribuir o utilizar el corpus para ningún otro propósito.

2 1. Introducción al procesamiento de señales temporales de voz

2.1 1.1. Descarga de ficheros de ejemplo

Primero vamos a descargar el audio de ejemplo de Moodle (**audio_sample.wav**) y ejecutar las siguientes líneas de código, que nos permitirán subir el archivo a Google Colab desde el disco local:

[ ]:
```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving audio_sample.wav to audio_sample.wav
```

Una vez cargado el fichero de audio, podemos escucharlo de la siguiente manera:

[ ]:
```
import IPython

wav_file_name = "audio_sample.wav"
print(wav_file_name)
IPython.display.Audio(wav_file_name)
```

```
audio_sample.wav
```

[ ]:
```
<IPython.lib.display.Audio object>
```

2.2 1.2. Lectura y representación de audio en Python

A continuación vamos a definir ciertas funciones para poder hacer manejo de ficheros de audio en Python.

Comenzamos definiendo una función **read_recording** que leerá un fichero de audio WAV, normalizará la amplitud y devolverá el vector de muestras *signal* y su frecuencia de muestreo *fs*.

[21]:
```
import scipy.io.wavfile

def read_recording(wav_file_name):
  fs, signal = scipy.io.wavfile.read(wav_file_name)
  signal = signal/max(abs(signal)) # normalizes amplitude

  return fs, signal
```

Si ejecutamos la función anterior para el fichero de ejemplo, podemos ver la forma en la que se carga dicho fichero de audio en Python. Así, podemos obtener la frecuencia de muestreo y la longitud del fichero en número de muestras:

[ ]:
```
fs, signal = read_recording(wav_file_name)
print("Signal variable shape: " + str(signal.shape))
print("Sample rate: " + str(fs))
print("File length: " + str(len(signal)) + " samples")
```

```
Signal variable shape: (4800160,)
Sample rate: 8000
File length: 4800160 samples
```

PREGUNTAS:

- ¿Como obtendría la duración de la señal en segundos?

RESPUESTA

Consulta la documentación oficial de scipiy vemos que la frecuencia de muestra es devuelta en **muestra/segundo**. Para obtener la duración de la señal en tiempo basta con dividir el número de muestra entre la frecuencia de muestreo:

[ ]:
```
total_time = 1.0 * len(signal) / fs
print("File duration (in seconds): {}".format(total_time))
```

```
File duration (in seconds): 4.192
```

También podemos representar la señal y ver su forma de onda. Para ello, definimos la función **plot_signal** como sigue:

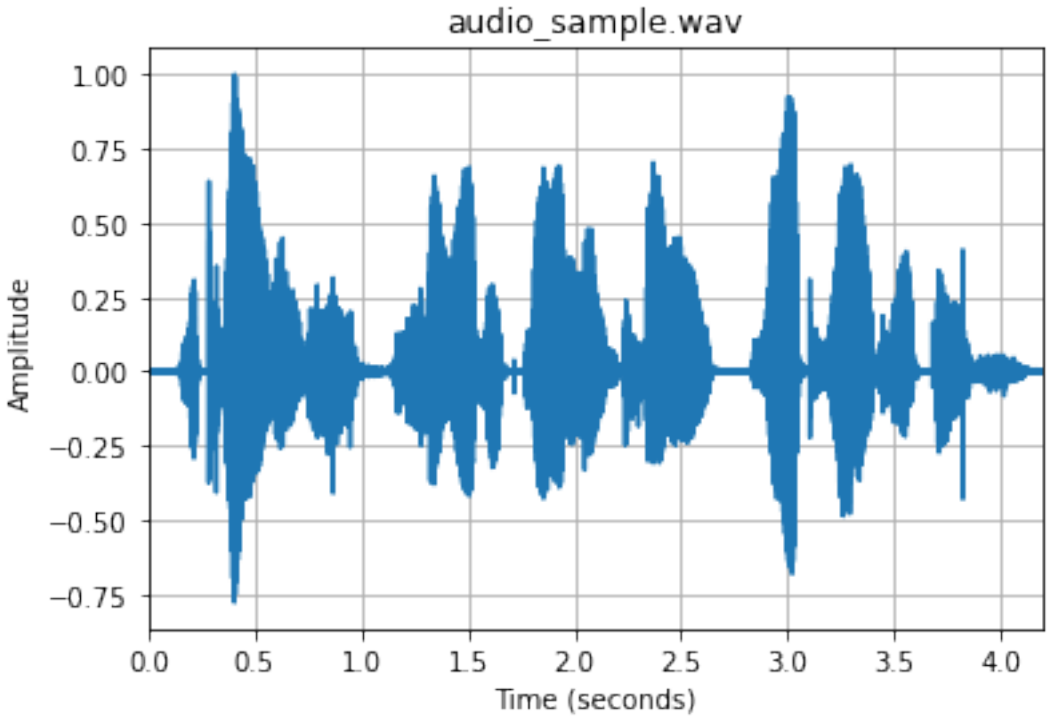
[24]:
```
import matplotlib.pyplot as plt
import numpy as np

def plot_signal(signal, fs, ylabel="", title=""):
  dur = len(signal)/fs
  step = 1./fs
  t_axis = np.arange(0., dur, step)

  plt.plot(t_axis, signal)
  plt.xlim([0, dur])
  plt.ylabel(ylabel)
  plt.xlabel('Time (seconds)')
  plt.title(title)
  plt.grid(True)
```

Y utilizando la función anterior, obtenemos su representación (amplitud frente al tiempo):

[ ]:
```
plot_signal(signal, fs, "Amplitude", wav_file_name)
plt.show()
```



PREGUNTAS:

- Incluya en el informe la representación obtenida.

RESPUESTA

Incluida.

2.3 1.3. Representación de etiquetas de actividad de voz

En esta práctica, vamos a desarrollar un detector de actividad de voz, que determinará qué segmentos de la señal de voz son realmente voz y cuáles silencio.

Por ello, vamos a ver dos ejemplos de etiquetas *ground truth*, que corresponden al fichero de audio de ejemplo.

Primero, descargamos de Moodle las etiquetas de voz/silencio que están en los ficheros **audio_sample_labels_1.voz** y **audio_sample_labels_2.voz** y las cargamos en Google Colab como en el caso anterior.

[ ]:
```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

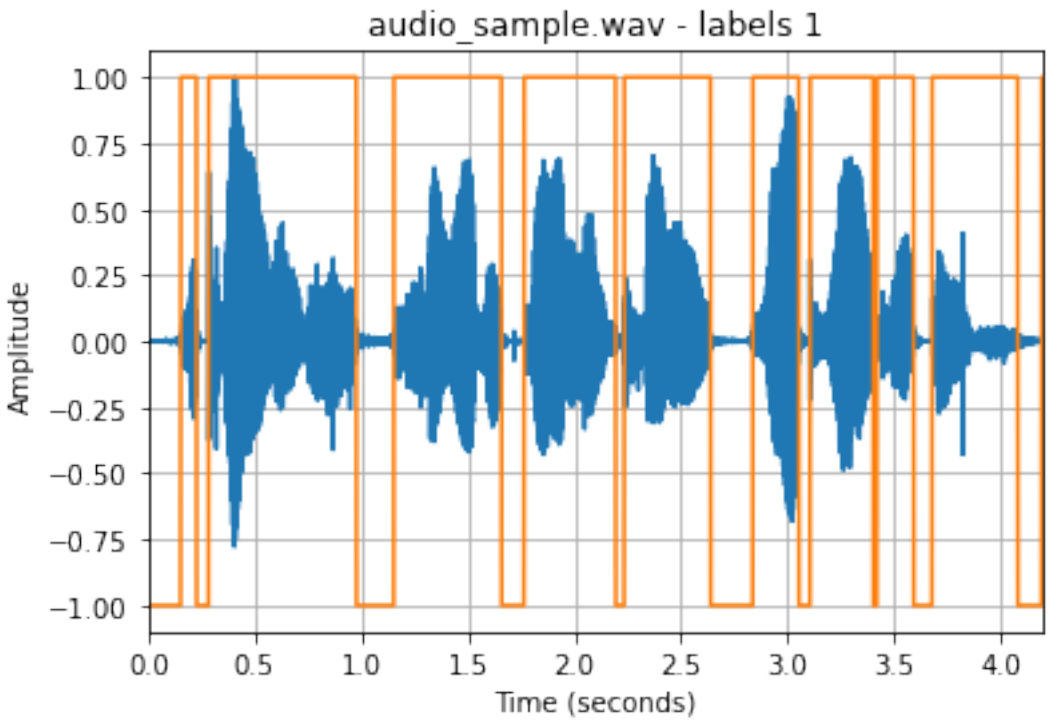
```
Saving audio_sample_labels_1.voz to audio_sample_labels_1.voz
Saving audio_sample_labels_2.voz to audio_sample_labels_2.voz
```

Estas etiquetas están guardadas en ficheros de texto y podemos cargarlas en Python de la siguiente manera:

[ ]:
```
labels_file_name = 'audio_sample_labels_1.voz'
voice_labels = np.loadtxt(labels_file_name)
```

Con el siguiente código, podemos representar la señal de voz así como sus etiquetas en la misma figura:

[ ]:
```
plot_signal(signal, fs)
plot_signal(voice_labels*2-1, fs, "Amplitude", '{} - labels 1'.
 ↪format(wav_file_name))
plt.show()
```



Las etiquetas de voz/silencio provienen de distintos detectores de actividad de voz.

PREGUNTAS: 1. ¿Qué valores tienen las etiquetas? ¿Qué significan dichos valores? 2. ¿Por qué se representa _voice_labels*2-1_? 3. Represente la señal de voz junto con las etiquetas para ambos casos e incluya las figuras en el informe de la práctica. ¿Qué diferencias observas? ¿A qué se puede deber? 4. ¿Qué cantidad de voz/silencio hay en cada etiquetado?

RESPUESTAS

1. Las etiquetas toman los valores 0 y 1 para simbolizar que no se está o si se está hablando, respectivamente

[ ]:
```
voice_labels
```

[ ]:
```
array([0., 0., 0., …, 1., 1., 1.])
```

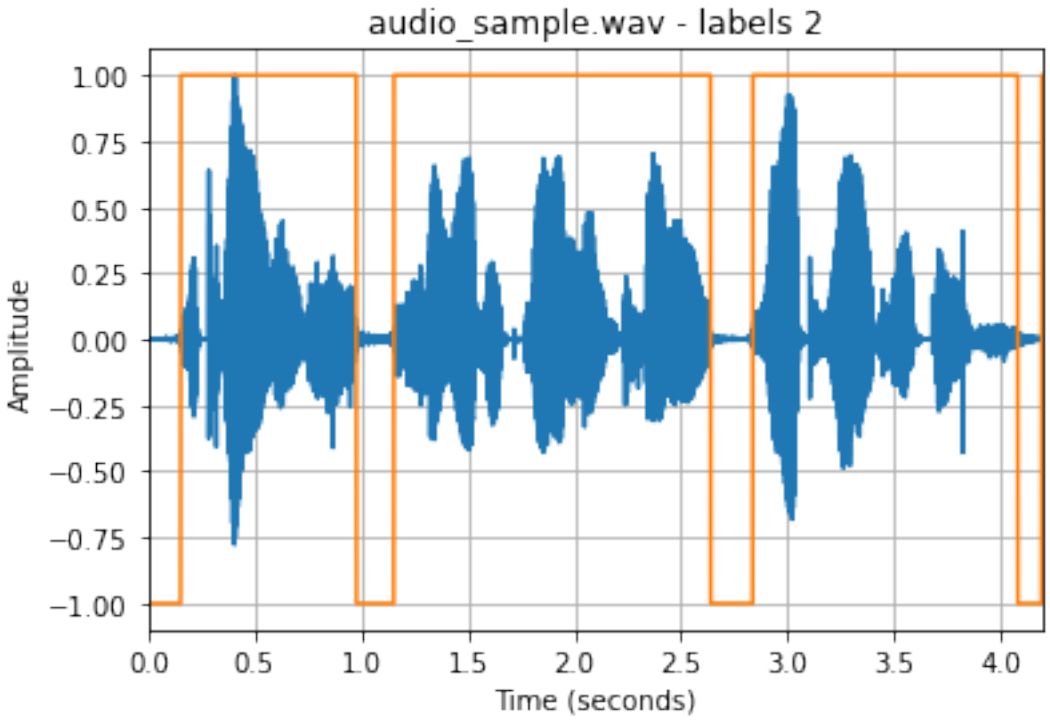
2. Representamos _voice_labels*2-1_ para mapear la señal al intervalo `[-1,1]` y poder visualizarla mejor en comparativa.

3. La primera gráfica está representada arriba. Para el segundo conjunto de etiquetas repetimos el proceso:

[ ]:
```
labels_file_name_2 = 'audio_sample_labels_2.voz'
voice_labels_2 = np.loadtxt(labels_file_name_2)

plot_signal(signal, fs)
```

```
plot_signal(voice_labels_2*2-1, fs, "Amplitude", '{} - labels 2'.
 ↪format(wav_file_name))
plt.show()
```



En el primer conjunto de etiquetas hay muchos más *escalones*. Es decir, detectamos muchos más tramos pequeños de voy no voz. En el segundo tenemos únicamente 3 grandes trozos de voz. Este se peude deber a que durante el etiquetado hemos marcado intervalos más grandes, y no detectamos pequeños variaciones en tiempo.

4. Para responder a la última preguntamos contamos el número de etiquetas de cada intervalo y hacemos una regla de tres utilizando el tiempo total para calcular el tiempo de voz y silencio en cada tramo:

[ ]:
```
def print_times(labels, total_time):
  voice = np.mean(labels)
  silence = np.mean(1 - labels)

  print('\tVoice time (s): {}'.format(voice * total_time))
  print('\tSilence time (s): {}'.format(silence * total_time))

print('Labels set 1:')
print_times(voice_labels, total_time)
print('Labels set 2:')
print_times(voice_labels_2, total_time)
```

```
Labels set 1:
        Voice time (s): 3.1969375
        Silence time (s): 0.9950625000000001
Labels set 2:
        Voice time (s): 3.5544375
        Silence time (s): 0.6375625
```

2.4 1.4. Extracción de características

En la mayoría de sistemas de reconocimiento de patrones, un primer paso es la extracción de características. Esto consiste, a grandes rasgos, en obtener una representación de los datos de entrada, que serán utilizados para un posterior modelado.

En nuestro caso, vamos pasar de la señal en crudo *"raw"* dada por las muestras (*signal*), a una secuencia de vectores de características que extraigan información a corto plazo de la misma y la representen. Esta sería la entrada a nuestro sistema de detección de voz basado en redes neuronales.

Para ver algunos ejemplos, vamos a utilizar la librería *librosa* (https://librosa.org/doc/latest/index.html).

Dentro de esta librería, tenemos funciones para extraer distintos tipos de características de la señal de voz, como por ejemplo el espectrograma en escala Mel (*melspectrogram*).

Estas características a corto plazo, se extraen en ventanas de unos pocos milisegundos con o sin solapamiento.

Un ejemplo sería el siguiente:

[ ]:
```
import librosa

mel_spec = librosa.feature.melspectrogram(
  signal,
  fs,
  n_mels=23,
  win_length=320,
  hop_length=160
)

print(mel_spec.shape)
print(signal.shape)
```

```
(23, 420)
(67072,)
```

PREGUNTAS: 1. ¿Qué se obtiene de la función anterior? 2. ¿Qué significan los valores de los parámetros *win_length* y *hop_length*? 3. ¿Qué dimensiones de *mel_spec* obtienes? ¿Qué significan?

RESPUESTAS

1. El espectograma en escala de *Mel*.
2. *win_length* especifica el tamaño de las ventanas en las que subdividiremos nuestra señal. El tamaño final de la ventana será *n_fft* (2048 por defecto), el resto se completa con *0-*

padding. Por otro lado, *hop_length* define el *tamaño de salto*, o cada cuantas muestras tomamos una neuva ventana. en nuestro caso, como `win_length = 2*hop_length`, tenemos un solapamiento entre ventanas del `50%`.

3. La primera dimensión hace referencia a la frencuencia. Cada fila representa los herzios encontrados para una frecuencia especifica a lo largo de toda la señal. Puesto que hemos fijado *n_mels=23*, obtenemos *23* posibles frecuencias.

La segunda dimensión referencia el tiempo. Como sabemos, nuestra señal dura aproximadamente 4.2 segundos, cada columna del melspectrograma representa los distintos valores de la señal para distintas frecuencias en un intervalo de tiempo fijo de tamaño 0.01 segundos.

Podemos saber de antemano el valor de t: Será el tamaño total de nuestra señal dividido entre el tamaño de salto redondeado hacia arriba:

[ ]:
```
67072 / 160
```

[ ]:
```
419.2
```

De esta manera, podríamos obtener una parametrización de las señales para ser utilizadas como entrada a nuestra red neuronal.

Para los siguientes apartados, se proporcionan los vectores de características MFCC para una serie de audios que se utilizarán como conjunto de entrenamiento del modelo de VAD.

#2. Detector de actividad de voz (Voice Activity Detector, VAD)

2.5 2.1. Descarga de los datos de entrenamiento

Primero vamos a descargar la lista de identificadores de los datos de entrenamiento de la práctica.

Para ello, necesitaremos descargar de Moodle el fichero **training_VAD.lst**, y ejecutar las siguientes líneas de código, que nos permitirán cargar el archivo a Google Colab desde el disco local:

[2]:
```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving data_download_onedrive_training_VAD.sh to
data_download_onedrive_training_VAD.sh
Saving data_download_onedrive_valid_VAD.sh to
data_download_onedrive_valid_VAD.sh
Saving training_VAD.lst to training_VAD.lst
Saving valid_VAD.lst to valid_VAD.lst
```

A continuación cargamos los identificadores contenidos en el fichero en una lista en Python:

[3]:
```
file_train_list = 'training_VAD.lst' # mat files containing data + labels
f = open(file_train_list, 'r')
train_list = f.read().splitlines()
f.close()
```

Podemos ver algunos de ellos (los primeros 10 identificatores) de la siguiente forma:

[4]:
```
print(train_list[:10])
```

```
['features_labs_1.mat', 'features_labs_10.mat', 'features_labs_100.mat',
'features_labs_101.mat', 'features_labs_102.mat', 'features_labs_103.mat',
'features_labs_104.mat', 'features_labs_105.mat', 'features_labs_106.mat',
'features_labs_107.mat']
```

Ahora, descargaremos de Moodle el fichero **data_download_onedrive_training_VAD.sh**, y ejecutaremos las siguientes líneas de código, que nos permitirán cargar el archivo a Google Colab desde el disco local:

[ ]:
```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>

Saving data_download_onedrive_training_VAD.sh to
data_download_onedrive_training_VAD.sh
```

Para descargar el conjunto de datos desde One drive, ejecutamos el script cargado anteriormente de la siguiente manera:

[5]:
```
# Si esta celda falla con un "bad interpreter error" usa:
!sed -i -e 's/\r$//' data_download_onedrive_training_VAD.sh
!chmod 755 -R data_download_onedrive_training_VAD.sh
!./data_download_onedrive_training_VAD.sh
```

```
--2022-03-10 20:04:31--  https://dauam-my.sharepoint.com/:u:/g/personal/alicia_l
ozano_uam_es/EdCueYU7BpNAuo6BawH8hJAB5rclap745BmsPzXgSPhsgw?download=1
Resolving dauam-my.sharepoint.com (dauam-my.sharepoint.com)… 13.107.136.9,
13.107.138.9
Connecting to dauam-my.sharepoint.com (dauam-
my.sharepoint.com)|13.107.136.9|:443… connected.
HTTP request sent, awaiting response… 302 Found
Location: /personal/alicia_lozano_uam_es/Documents/PIT/training_VAD.zip
[following]
--2022-03-10 20:04:32--  https://dauam-
my.sharepoint.com/personal/alicia_lozano_uam_es/Documents/PIT/training_VAD.zip
Reusing existing connection to dauam-my.sharepoint.com:443.
HTTP request sent, awaiting response… 200 OK
Length: 3638232935 (3.4G) [application/x-zip-compressed]
Saving to: ‘EdCueYU7BpNAuo6BawH8hJAB5rclap745BmsPzXgSPhsgw?download=1’

EdCueYU7BpNAuo6BawH 100%[===================>]   3.39G  47.1MB/s    in 78s

2022-03-10 20:05:50 (44.4 MB/s) -
‘EdCueYU7BpNAuo6BawH8hJAB5rclap745BmsPzXgSPhsgw?download=1’ saved
[3638232935/3638232935]
```

Este script descargará los datos de One Drive y los cargará en Google Colab, descomprimiéndolos en la carpeta **data/training_VAD**.

Podemos comprobar que los ficheros **.mat** se encuentran en el directorio esperado:

[6]:
```
!ls data/training_VAD/ | head
```

```
features_labs_100.mat
features_labs_101.mat
features_labs_102.mat
features_labs_103.mat
features_labs_104.mat
features_labs_105.mat
features_labs_106.mat
features_labs_107.mat
features_labs_108.mat
features_labs_109.mat
```

2.6 2.2. Definición del modelo

Utilizando la librería Pytorch (https://pytorch.org/docs/stable/index.html), vamos a definir un modelo de ejemplo con una capa LSTM y una capa de salida. La capa de salida estará formada por una única neurona. La salida indicará la probabilidad de voz/silencio utilizando una función *sigmoid*.

[7]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Model_1(nn.Module):
    def __init__(self, feat_dim=20):
        super(Model_1, self).__init__()

        self.lstm = nn.LSTM(feat_dim,256,batch_first=True,bidirectional=False)
        self.output = nn.Linear(256,1)

    def forward(self, x):

        out = self.lstm(x)[0]
        out = self.output(out)
        out = torch.sigmoid(out)

        return out.squeeze(-1)
```

PREGUNTAS: 1. ¿Qué tamaño tiene la entrada a la capa LSTM?

Queda fijado por el primer parámetro pasado a la capa LSTM: `input_size=fet_dim`, 20 por defecto en neustroc aso.

2. ¿Cuántas unidades (celdas) tiene dicha capa LSTM?

Queda fijado por el segundo parámetro, `hidden_size=256` en nuestro caso.

3. ¿Qué tipo de matriz espera la LSTM? Mirar la documentación y describir brevemente.

Como hemos fijado `batch_first=True`, espera una matriz de la forma (N, L, H_{in}); donde N es el tamaño de batch, L es el tamaño de secuencia y H_{in} el tamaño de la entrada.

4. Revisar la documentación de *torch.nn.LSTM* y describir brevemente los argumentos *batch_first*, *bidirectional* y *dropout*.

- *batch_first* invierte las primeras dos dimensiones de la matriz de entrada (pasa de ser (L, N, H_{in}) a (N, L, H_{in})).
- *bidirectional* decide si la red será bidireccional o no.
- *dropout*, si es mayor que 0 añade una capa de *dropout* tras cada capa de la LSTM con probabilidad el valor de *dropout* pasado por parámetro.

5. En este modelo, estamos utilizando una única neurona a la salida. ¿Hay alguna otra alternativa? ¿Se seguiría utilizando una función *sigmoid*?

Utilizamos una única neurona de salida que codifica la probabilidad de que haya o no voz en ese momento. Podríamos usar dos neuronas, una que codifique la probabilidad de que sea voz y otra que sea la probabilidad de codificar silencio. En este caso utilizaríamos dos sigmoides, una para cada neurona de salida.

Para computar dicha neurona de salida utilizamos ahora mismo un modelo lineal. De cara a computar dos neuronas podríamos mantener el modelo lineal para cada una de ellas, o bien introducir modelos más complejos, como un MLP.

6. ¿Para qué sirve la función *forward* definida en la clase *Model_1*?

La función *foward* computa la salida de la red a partir de la entrada x.

Una vez definida la clase, podemos crear nuestra instancia del modelo y cargarlo en la GPU con el siguiente código:

[8]:
```
model = Model_1(feat_dim=20)
model = model.to(torch.device("cuda"))
print(model)
```

```
Model_1(
  (lstm): LSTM(20, 256, batch_first=True)
  (output): Linear(in_features=256, out_features=1, bias=True)
)
```

Nuestra variable *model* contiene el modelo, y ya estamos listos para entrenarlo y evaluarlo.

##2.3. Lectura y preparación de los datos para el entrenamiento

Como hemos visto anteriormente, nuestros datos están guardados en ficheros de Matlab (**.mat**). Cada uno de estos ficheros contiene una matriz **X** correspondiente a las secuencias de características MFCC (con sus derivadas de primer y segundo orden), y un vector **Y** con las etiquetas de voz/silencio correspondientes.

Veamos un ejemplo:

[9]:
```
features_file = 'data/training_VAD/features_labs_1.mat'

import scipy.io
features = scipy.io.loadmat(features_file)['X']
labels = scipy.io.loadmat(features_file)['Y']

print(features.shape)
print(labels.shape)
```

```
(46654, 60)
(46654, 1)
```

PREGUNTAS: Elegir un fichero de entrenamiento y responder a las siguientes preguntas: 1. ¿Qué tamaño tiene **features**? ¿Y **labels**?

En el archivo elegido tenemos 46654 instantes temporales con 60 características cada una. El tamaño de labels es justamente el número de instantes temporales, 46654, que indican el valor de la señal en dicho instante.

2. Una de las dimensiones de la **features** es 60, correspondiente a los 20 coeficientes MFCC concatenados con las derivadas de primer y segundo orden. ¿Con qué se corresponde la otra dimensión?

Como comentábamos, es el número de ventanas tomadas al enventanar la señal.

El entrenamiento del modelo se va a realizar mediante descenso por gradiente (o alguna de sus variantes) basado en *batches*.

Para preparar cada uno de estos *batches* que servirán de entrada a nuestro modelo LSTM, debemos almacenar las características en secuencias de la misma longitud. El siguiente código lee las características (**get_fea**) y sus correspondientes etiquetas (**get_lab**) de un fragmento aleatorio del fichero de entrada.

[10]:
```
import scipy.io
import numpy as np

length_segments = 300

def get_fea(segment, rand_idx):
    data = scipy.io.loadmat(segment)['X']
    if data.shape[0] <= length_segments:
        start_frame = 0
    else:
        start_frame = np.random.permutation(data.shape[0]-length_segments)[0]

    end_frame = np.min((start_frame + length_segments,data.shape[0]))
    rand_idx[segment] = start_frame
    feat = data[start_frame:end_frame,:20] # discard D and DD, just 20 MFCCs
    return feat[np.newaxis, :, :]
```

```
def get_lab(segment, rand_idx):
    data = scipy.io.loadmat(segment)['Y']
    start_frame = rand_idx[segment]
    end_frame = np.min((start_frame + length_segments, data.shape[0]))
    labs = data[start_frame:end_frame].flatten()
    return labs[np.newaxis,:]
```

PREGUNTAS: Analizar las funciones anteriores detenidamente y responder a las siguientes cuestiones:

- ¿De qué tamaño son los fragmentos que se están leyendo?

De tamaño *length_segments*, 300 por defecto.

- ¿Para qué sirve *rand_idx*?

Es el diccionario de indices (potencialmente aleatorios) donde empieza los segmentos aleatorios a recortar.

2.7 2.4. Entrenamiento del modelo

Una vez definidas las funciones de lectura de datos y preparación del formato que necesitamos para la entrada a la red LSTM, podemos utilizar el siguiente código para entrenarlo.

[14]:
```
from torch import optim

def compute_acc(model, segments_sets, path_in_feat, portion_of_data=None):
  acc = 0
  total_labels = 0

  # Compute the percentage of data to use
  portion = portion_of_data if portion_of_data is not None else 1
  n_max = int( portion * len(segments_sets))

  for _, segment_set in enumerate(segments_sets):
    rand_idx = {}
    batch = np.vstack([get_fea(path_in_feat + segment, rand_idx) for segment in␣
↪segment_set])
    labs_batch = np.vstack([get_lab(path_in_feat + segment, rand_idx).astype(np.
↪int16) for segment in segment_set])

    # Tensor data
    batch = torch.tensor(batch.take(shuffle, axis=0).astype("float32")).
↪to(torch.device("cuda"))
    batch_labels = torch.tensor(labs_batch.take(shuffle, axis=0).
↪astype("float32")).to(torch.device("cuda"))

    # Compute predictions
    outputs = model.forward(batch)
```

```
    outputs = torch.round(outputs)

    # Accumulate accuracy
    total_labels += len(outputs) * len(outputs[0])
    acc += torch.sum(outputs == batch_labels)
  # Compute the total accuracy
  return acc / total_labels
```

[16]:
```
#fix seeds
np.random.seed(123)
torch.manual_seed(123)

path_in_feat = 'data/training_VAD/'

criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

batch_size = 51
segment_sets = np.array_split(train_list, len(train_list)/batch_size)

max_iters = 5
for epoch in range(1, max_iters):
  print('Epoch: ', epoch)
  model.train()
  cache_loss = 0

  for ii, segment_set in enumerate(segment_sets):
    rand_idx = {}
    optimizer.zero_grad()

    # Create training batches
    train_batch = np.vstack([get_fea(path_in_feat + segment, rand_idx) for␣
↪segment in segment_set])
    labs_batch = np.vstack([get_lab(path_in_feat + segment, rand_idx).astype(np.
↪int16)  for segment in segment_set])
    assert len(labs_batch) == len(train_batch) # make sure that all frames have␣
↪defined label

    # Shuffle the data and place them into Pytorch tensors
    shuffle = np.random.permutation(len(labs_batch))
    labs_batch = torch.tensor(labs_batch.take(shuffle, axis=0).
↪astype("float32")).to(torch.device("cuda"))
    train_batch = torch.tensor(train_batch.take(shuffle, axis=0).
↪astype("float32")).to(torch.device("cuda"))

    # Forward the data through the network
    outputs = model(train_batch)
```

```
    # Compute cost
    loss = criterion(outputs, labs_batch)

    # Backward step
    loss.backward()
    optimizer.step()
    cache_loss += loss.item()

  # Change the model to evaluation mode
  model.eval()
  with torch.no_grad():
    # Compute accuracy values
    train_acc = compute_acc(model, segment_sets, path_in_feat, 0.5)

  # Print accuracy
  print("Acc: {}".format(train_acc))
  print("Loss: " + str(cache_loss/len(train_batch)))
```

```
Epoch:  1
Acc: 0.8611633777618408
Loss: 0.08033171062375985
Epoch:  2
Acc: 0.8756666779518127
Loss: 0.06632960485477074
Epoch:  3
Acc: 0.8795032501220703
Loss: 0.06001589461868884
Epoch:  4
Acc: 0.8788823485374451
Loss: 0.057638293972202374
```

PREGUNTAS: Analizar el código anterior cuidadosamente y ejecutarlo. A continuación, responder a las siguientes cuestiones: - ¿Qué función de coste se está optimizando? Describir brevemente con ayuda de la documentación.

La función de coste queda encapsulada en el objeto `criterion` y en nuestro caso es la **Entropía Cruzada Binaria**. Por defecto tomamos el valor medio sobre nuestras muestras. Se puede consultar la documentación oficial aquí.

- ¿Qué optimizador se ha definido?

Se ha utilizado `Adam` con *learning rate* `lr=0.001`.

- ¿Para qué se utiliza *batch_size*?

Para definir el tamaño de batch. Particionamos nuestro dataset en batches de este tamaño para procesarlos.

- Describir brevemente la creación de los *batches*.

Se divide el número de muestras entre el número de batches y se utiliza la función `np.array_split` para dividir en ese número de batches.

- ¿Qué línea de código realiza el *forward pass*?

La línea `32` computa las salidas de la red a partir de la entrada.

- ¿Qué línea de código realiza el *backward pass*?

La línea `38` computa los gradientes. Consultar la documentación oficial del método `backward()` para más información.

- ¿Cuántas iteraciones del algoritmo ha realizado? ¿Qué observa en la evolución de la función de coste?

Hemos realizado únicamente 4 iteraciones y vemos como la función de coste se va reduciendo en cada una, aunque en menor medida en las últimas.

- Añada al código el cálculo de la precisión o *accuracy*, de tal manera que se muestre por pantalla dicho valor en cada iteración (similar a lo que ocurre con el valor del coste *loss*). Copiar el código en el informe y describir brevemente.

He añadido al código una función que computa la precisión a partir del modelo, los segmentos, y el path al archivo que contiene los archivos a evaluar. Esto nos servirá posteriormente para realizar validación sobre nuestro modelo.

Tras varias iteraciones sobre cómo estudiar la precisión he decidido que estudiarla en un único segmento al final de cada época puede ser poco representativo. De la misma forma, no podemos evaluar durante la época en si pues no obtenemos valores representativos de cómo acaba el modelo al final de la misma.

Es por ello que la función que computa la precisión realiza un proceso análogo al realizado en entrenamiento: vuelve a cargar los archivos en batches, selecciona segmentos aleatorios de los mismos y realiza predicción sobre ellos.

Sobre el cómputo de la precisión en si caben destacar un par de detalles:

1. Las predicciones del modelo son valores probabilísticos (entre 0 y 1). Se han redondeado a 0/1 para realizar la comparativa final.
2. En la etapa de predicción del modelo podemos ahorrar memoria no computando los gradientes ejecutando `torch.no_grad()`, así cómo poner el modelo en modo predictivo para una mejor precisión (`model.eval()`).
3. Hemos fijado la semilla para el experimento completo para que sea reproducible.

- ¿Qué valor de coste y *accuracy* obtiene? ¿Cómo se puede mejorar?

Se obtienen los siguientes valores:

	Acc	Loss
1	0.8611	0.0803
2	0.8756	0.0663
3	0.8795	0.0600
4	0.8788	0.0576

Como podemos ver, no por entrenar más épocas estamos obteniendo resultados notablmente mayores, la mejora es lenta pero constante. Esto se debe a que estamos entrando sobre muchas tramas cortas de audio en cada época, y con una es casi suficiente para que el modelo converja. Aún así observamos una mínima mejoría con el entrenamiento progresivo.

De cara a mejorar estos resultados podríamos o bien mejorar nuestro modelo actual o bien probar con modelos más avanzados:

- Para mejorar el modelo actual podríamos añadir más capas ocultas, más LSTMs en nuestra capa oculta actual, convertir el clasificador final en algo más complejo que un clasificador lineal, utilizar dropout…
- De cara a probar modelos más complejos podríamos introducir técnicas más sofisticadas, como el uso de redes bidireccionales con LSTMs; también podríamos extraer características relevantes con las que alimentar nuestra red.

2.8 2.5. Evaluación del modelo: un único fichero de test

Una vez entrenado el modelo, vamos a evaluarlo en un ejemplo en concreto.

Descargue de Moodle el fichero **audio_sample_test.wav**, con sus correspondientes características y etiquetas **audio_sample_test.mat** y evalúe el rendimiento en el mismo.

2.8.1 Respuesta

Comenzamos cargando los ficheros:

[17]:
```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

```
Saving audio_sample_test.mat to audio_sample_test.mat
Saving audio_sample_test.wav to audio_sample_test.wav
```

Preparamos los datos de evaluación:

[18]:
```
features_file = 'audio_sample_test.mat'

import scipy.io
features = scipy.io.loadmat(features_file)['X']
labels = scipy.io.loadmat(features_file)['Y']

print(features.shape)
print(labels.shape)
```

```
(57777, 60)
(57777, 1)
```

Nuestro objetivo en este ejercicio será obtener una predicción generada por nuestro modelo para cada trama de audio. Para ello tomamos las 20 primeras posiciones de las features (evitando así las

derivadas, no necesarias para nuestras LSTMs), pasamos las features a CUDA y las alimentamos a nuestro modelo.

Finalmente, para computar la precisión hemos de sacar de CUDA el array de outputs obtenido.

[19]:
```
# Get only first 30 features, skipping derivatives
features = features[:,:20]

# Move features to tensor
features_tensored = torch.tensor([features]).to(torch.device("cuda")).float()

# Forward the data through the network
model.eval()
with torch.no_grad():
  outputs = model.forward(features_tensored)

# Convert the output from cuda tensor to numpy array
outputs = outputs.cpu().detach().numpy().flatten()

# Flatten the labels to compare them
labels = labels.flatten()

# Compute accuracy
outputs_rounded = np.round(outputs)
total_labels = len(labels)
test_acc = np.sum(outputs_rounded == labels) / total_labels
print('Test acc: {}'.format(test_acc))
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5: UserWarning:
Creating a tensor from a list of numpy.ndarrays is extremely slow. Please
consider converting the list to a single numpy.ndarray with numpy.array() before
converting to a tensor. (Triggered internally at
../torch/csrc/utils/tensor_new.cpp:201.)
  """

Test acc: 0.9066583588625231
```

PREGUNTAS: 1. Incluya en el informe de la práctica el código que ha utilizado para evaluar dicho fichero.

Incluido.

2. ¿Cuál es el *accuracy* obtenido para el fichero **audio_sample_test**?

Es aproximadamente 0.9125.

3. Represente 10 segundos de dicho audio, así como sus etiquetas de *ground_truth* y las obtenidas con su modelo. Incluya dicha gráfica en el informe y comente brevemente el resultado. Visualmente, ¿es bueno el modelo?

[22]:
```
# Read teh audio file
wav_file_name = 'audio_sample_test.wav'
fs, signal = read_recording(wav_file_name)

print("Signal variable shape: " + str(signal.shape))
print("Sample rate: " + str(fs))
```

```
Signal variable shape: (4800160,)
Sample rate: 8000
```

Para poder representar la señal junto a las labels nos damos cuenta de que han sido muestreadas con distintas frecuencia. Para la señal sabemos que la frecuencia es de 8000 por la celda anterior.

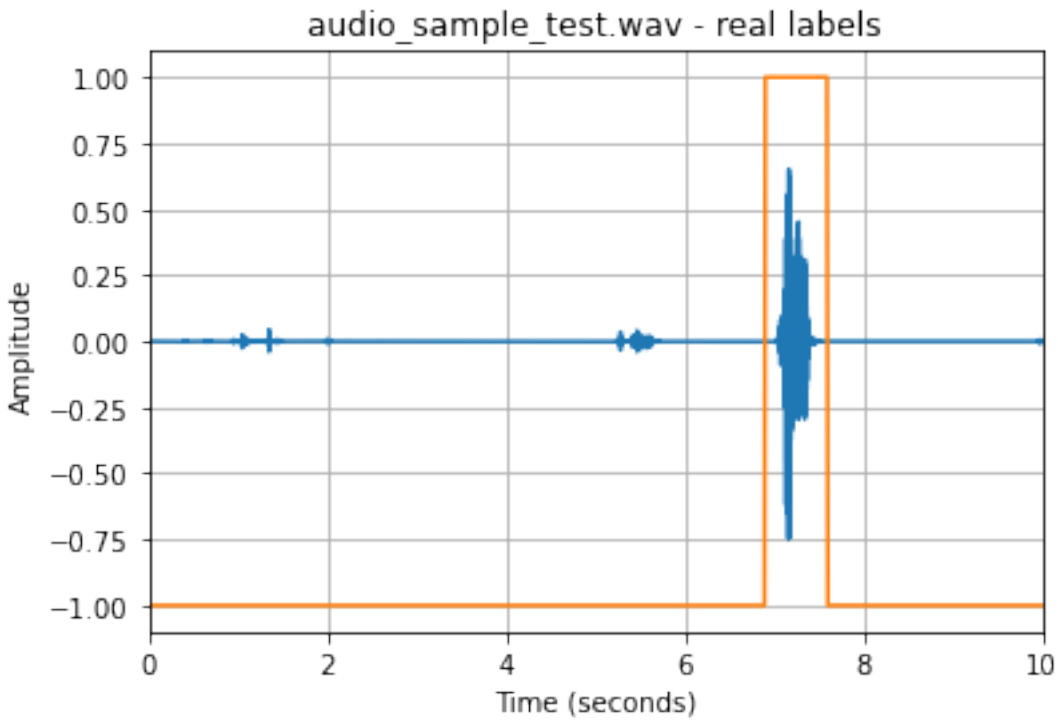
Para las etiquetas, sabemos que cada etiqueta corresponde a 0.01 segundos (esto es, el periodo T es de 0.01 segundos), así que la frecuencia será $1/T = 100$. Finalmente hemos de computar también el rango de las etiquetas manualmente.

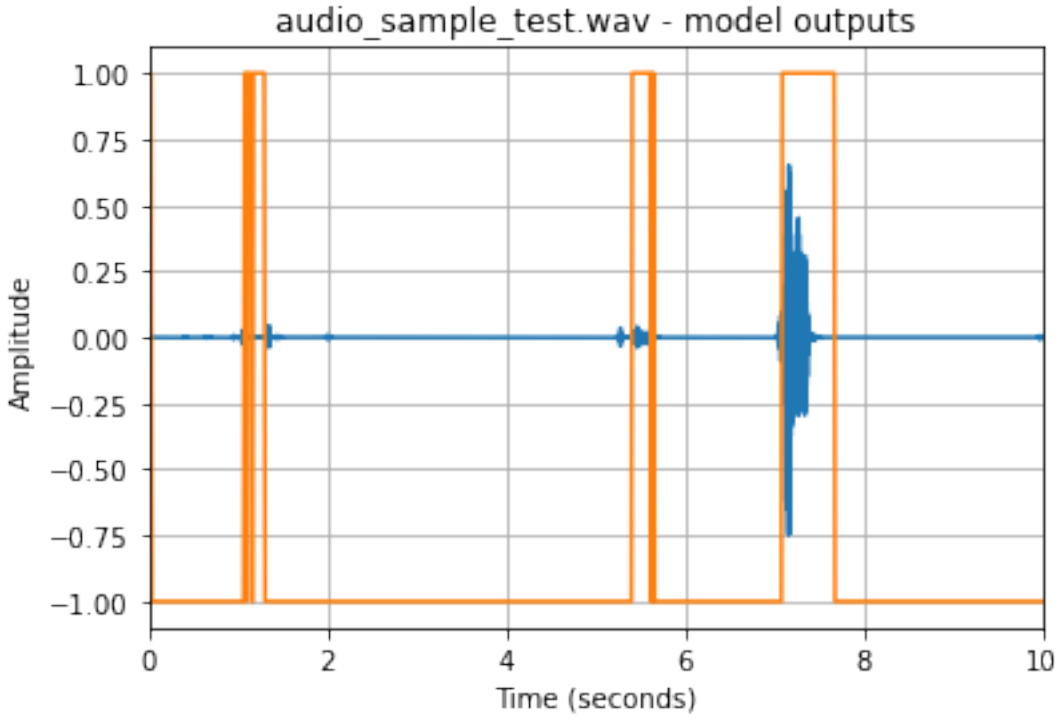
[25]:
```
# Grab only the first ten seconds of audio. We would need 10*f_s samples
n_samples_10_s = int(fs * 10)
signal_cropped = signal[:n_samples_10_s]

# Grab the labels for only the first 10s of audio
T_labels = 0.01
fs_labels = 1 / T_labels
labels_x = np.linspace(0, 10, num=int(10/T_labels))
labels_cropped = np.array(labels[:len(labels_x)], dtype=int)
outputs_rounded = np.array(outputs_rounded[:len(labels_x)], dtype=int)

plot_signal(signal_cropped, fs)
plot_signal(labels_cropped*2-1, fs_labels, "Amplitude", '{} - real labels'.
 ↪format(wav_file_name))
plt.show()

plot_signal(signal_cropped, fs)
plot_signal(outputs_rounded*2-1, fs_labels, "Amplitude", '{} - model outputs'.
 ↪format(wav_file_name))
```


audio_sample_test.wav - real labels
Amplitude
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
Time (seconds)


audio_sample_test.wav - model outputs
Amplitude
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2
4
6
8
10
Time (seconds)

El modelo detecta la parte principal de voz de nuestro audio casi en su totalidad pero le cuesta acertar el comienzo del mismo. Es normal, ya que hasta esa momento una red de LSTMs unicamente tiene el contexto previo al isntante actual.

De la misma forma, en un instante intermedio en el que no hay voz pero si una ligera perturbación nuestro modelo detecta que va a empezar a haber voz, pero acaba retractándose.

Creo que ambos casos se solucionarían con una red de LSTMs bidireccional, donde tenemos información también del futuro y podemos detectar mejor tanto los falsos positivos como el momento anterior a empezar a hablar.

4. Escuche el audio y comente cualitativamente cómo es de bueno o malo el modelo.

[26]:
```
import IPython

wav_file_name = "audio_sample_test.wav"
print(wav_file_name)
IPython.display.Audio(wav_file_name)
```

```
audio_sample_test.wav
```

[26]:
```
<IPython.lib.display.Audio object>
```

Escuchando los primeros 10 segundos de audio percibimos un poco de ruido antes del segundo 7, correspondiente a las pequeñas perturbaciones observadas en la señal. A continuación se escucha un claro `hello`, correspondiente al único momento con verdadera voz de nuestra trama de 10 segundos.

Aunque en el audio hay ruido ligero, es sencillo apreciar la voz para nosotros. En los primeros 10 segundos, el modelo se comporta de forma bastante efectiva acertando en la mayor parte del audio.

2.9 2.6. Evaluación del modelo: conjunto de validación

Ahora vamos a evaluar el rendimiento del modelo anterior sobre un conjunto de validación (del que conocemos sus etiquetas).

Para este conjunto de datos, descargaremos la lista de identificadores **valid_VAD.lst** de Moodle, así como el fichero de descarga de datos **data_download_onedrive_valid_VAD.sh**:

[27]:
```
from google.colab import files
uploaded = files.upload()
```

```
<IPython.core.display.HTML object>
```

[29]:
```
!sed -i -e 's/\r$//' data_download_onedrive_valid_VAD.sh
!chmod 755 data_download_onedrive_valid_VAD.sh
!./data_download_onedrive_valid_VAD.sh
```

```
mkdir: cannot create directory ‘./data’: File exists
--2022-03-10 20:36:08--  https://dauam-my.sharepoint.com/:u:/g/personal/alicia_l
ozano_uam_es/EWBjWyX774pLhJc2ahr4zk0BtLvWt7YGhdMDDmGu-LcBNQ?download=1
Resolving dauam-my.sharepoint.com (dauam-my.sharepoint.com)… 40.108.241.27
```

```
Connecting to dauam-my.sharepoint.com (dauam-
my.sharepoint.com)|40.108.241.27|:443… connected.
HTTP request sent, awaiting response… 302 Found
Location: /personal/alicia_lozano_uam_es/Documents/PIT/valid_VAD.zip [following]
--2022-03-10 20:36:09--  https://dauam-
my.sharepoint.com/personal/alicia_lozano_uam_es/Documents/PIT/valid_VAD.zip
Reusing existing connection to dauam-my.sharepoint.com:443.
HTTP request sent, awaiting response… 200 OK
Length: 508492067 (485M) [application/x-zip-compressed]
Saving to: 'EWBjWyX774pLhJc2ahr4zk0BtLvWt7YGhdMDDmGu-LcBNQ?download=1'

EWBjWyX774pLhJc2ahr 100%[===================>] 484.94M  22.3MB/s    in 23s

2022-03-10 20:36:32 (21.1 MB/s) - 'EWBjWyX774pLhJc2ahr4zk0BtLvWt7YGhdMDDmGu-
LcBNQ?download=1' saved [508492067/508492067]
```

[30]:
```
file_val_list = 'valid_VAD.lst' # mat files containing data + labels
f = open(file_val_list, 'r')
val_list = f.read().splitlines()
f.close()
```

Escriba ahora el código necesario para evaluar el modelo anterior en el conjunto de datos de validación, para su última época.

Tenga en cuenta que si quiere realizar el forward para todos los datos de validación de una vez, necesitará que todas las secuencias sean de la misma longitud. Como aproximación, puede escoger unos pocos segundos de cada fichero como se hace en el entrenamiento.

[31]:
```
def train_and_validate_model(model, model_name, seed=123):
  print('--- Model {} ---'.format(model_name))
  # Fix seeds
  np.random.seed(seed)
  torch.manual_seed(seed)

  path_in_feat = 'data/training_VAD/'
  path_in_feat_val = 'data/valid_VAD/'

  criterion = nn.BCELoss()
  optimizer = optim.Adam(model.parameters(), lr=0.001)

  batch_size = 51
  segment_sets = np.array_split(train_list, len(train_list)/batch_size)
  segment_sets_val = np.array_split(val_list, len(val_list)/batch_size)

  max_iters = 5
  for epoch in range(1, max_iters):
    print('Epoch: ', epoch)
```

```
  model.train()
  cache_loss = 0

  # --- TRAIN MODEL ---
  for _, segment_set in enumerate(segment_sets):
    rand_idx = {}
    optimizer.zero_grad()

    # Create training batches
    train_batch = np.vstack([get_fea(path_in_feat + segment, rand_idx) for␣
↪segment in segment_set])
    labs_batch = np.vstack([get_lab(path_in_feat + segment, rand_idx).
↪astype(np.int16)  for segment in segment_set])
    assert len(labs_batch) == len(train_batch) # make sure that all frames␣
↪have defined label

    # Shuffle the data and place them into Pytorch tensors
    shuffle = np.random.permutation(len(labs_batch))
    labs_batch = torch.tensor(labs_batch.take(shuffle, axis=0).
↪astype("float32")).to(torch.device("cuda"))
    train_batch = torch.tensor(train_batch.take(shuffle, axis=0).
↪astype("float32")).to(torch.device("cuda"))

    # Forward the data through the network
    outputs = model(train_batch)

    # Compute cost
    loss = criterion(outputs, labs_batch)

    # Backward step
    loss.backward()
    optimizer.step()
    cache_loss += loss.item()

  # Change the model to evaluation mode
  model.eval()

  with torch.no_grad():
    # Compute train accuracy
    train_acc = compute_acc(model, segment_sets, path_in_feat)

    # Compute test accuracy
    val_acc = compute_acc(model, segment_sets_val, path_in_feat_val)

  # --- Print metrics ---
  print("Loss: " + str(cache_loss/len(train_batch)))
  print("Train acc: {}".format(train_acc))
```

```
    print("Val acc: {}".format(val_acc))

train_and_validate_model(model, model_name='1')
```

```
--- Model 1 ---
Epoch:  1
Loss: 0.05622965857094409
Train acc: 0.8880849480628967
Val acc: 0.8838561773300171
Epoch:  2
Loss: 0.06083275932891696
Train acc: 0.8857581615447998
Val acc: 0.8706535696983337
Epoch:  3
Loss: 0.057405272535249297
Train acc: 0.8930914998054504
Val acc: 0.9064705967903137
Epoch:  4
Loss: 0.05343762069356208
Train acc: 0.9028888940811157
Val acc: 0.9012418389320374
```

PREGUNTAS: 1. Incluya en la memoria de la práctica el código utilizado, incluyendo los valores de cualquier parámetro de configuración utilizado (por ejemplo, el número de épocas de entrenamiento realizadas).

El código se ha incluido arriba, donde pueden apreciarse todos los parámetros. El número de épocas ha sido fijado a `4`, el tamaño de batch a `51` y la semilla a `123`.

Cabe destacar que todo el cómputo de entrenamiento y predicción de nuestro modelo en training y test se ha encapsulado en una función para su posterior reutilización.

2. ¿Qué rendimiento (loss y accuracy) obtiene con este modelo (*Model_1*) en entrenamiento y en validación?

	Acc	Val Acc	Loss
1	0.8881	0.8839	0.0562
2	0.8858	0.8707	0.0608
3	0.8931	0.9065	0.0574
4	0.9029	0.9012	0.0576

Observamos valores ligeramente distintos a los anteriores a pesar de haber fijado la semilla. Esto se debe a que ahora generamos más números aleatorios: al final de cada época estamos evaluando tanto en los datos de training como en los de validación, y para ambos casos generamos números aleatorios.

Podemos apreciar en la tabla que los valores de precisión y precisión en validación no distan tanto entre si. De hecho, tras la tercera época el valor de precisión en validación es superior al de entrenamiento. Esto podría indicarnos que estamos generalizando bien con nuestro modelo y ni se

acerca a producirse *overfitting*.

3 3. Comparación de modelos

3.1 3.1. Redes LSTM bidireccionales

En este apartado, vamos a partir del modelo inicial (*Model_1*) y modificarlo para que la capa LSTM sea bidireccional (*Model_1B*).

Entrene el nuevo modelo y compare el resultado con el modelo inicial.

[37]:
```
class Model_1B(nn.Module):
  def __init__(self, feat_dim=20):
    super(Model_1B, self).__init__()

    # Define the LSTM layer as bidirectional
    self.lstm = nn.LSTM(feat_dim, 256, batch_first=True, bidirectional=True)

    # Change the number of input links from 256 to 512
    self.output = nn.Linear(512,1)

  def forward(self, x):

    out = self.lstm(x)[0]
    out = self.output(out)
    out = torch.sigmoid(out)

    return out.squeeze(-1)
```

[39]:
```
# Define de model
model = Model_1B(feat_dim=20)
model = model.to(torch.device("cuda"))

# Train and validate it
train_and_validate_model(model, model_name='1B')
```

```
--- Model 1B ---
Epoch:  1
Loss: 0.11214799273247812
Train acc: 0.7998039126396179
Val acc: 0.7847058773040771
Epoch:  2
Loss: 0.08524513127757054
Train acc: 0.8642222285270691
Val acc: 0.8229411840438843
Epoch:  3
Loss: 0.05944389747638328
Train acc: 0.8955163359642029
```

```
Val acc: 0.9092810153961182
Epoch:  4
Loss: 0.05427969320147645
Train acc: 0.8937320113182068
Val acc: 0.9117646813392639
```

PREGUNTAS: 1. Explique brevemente la diferencia entre una capa LSTM y una BLSTM (bidirectional LSTM).

Una capa LTM bidireccional incluye dos capas ocultas que transmiten la informaciónde forma recurrente hacia delante y hacia atrás temporalmente respectivamente.

2. Incluya el código donde define *Model_1B* en el informe de la práctica.

Incluido arriba. Los dos únicos detalles relevantes son la línea `6` donde incluímos que la capa sea bidireccional y cambiar la entrada del clasificador lineal en la línea `9` para que acepta el doble de neuronas (de las dos capas que componen la capa bidireccional).

3. ¿Qué modelo obtiene un mejor resultado sobre los datos de validación? ¿Por qué puede ocurrir esto?

	Acc	Val Acc	Loss
1	0.7998	0.7847	0.1121
2	0.8642	0.8229	0.0852
3	0.8955	0.9093	0.0594
4	0.8937	0.9118	0.0543

Vemos que los valores obtenidos no se diferencian demasiado de los del modelo anterior. A pesar de ello sabemos que este tipo de red ha de entrenar mejor pues tiene información extra sobre el contexto, especialmente valiosa en este problema por lo que vimos en el apartado **2.5** cuando pintamos nuestra señal.

Mi teoría actual es que el modelo no ha convergido del todo. Entrenando con únicamente 4 épocas apenas pasamos sobre la cantidad ingente de datos que tenemos.

Al tener esta red dos capas ocultas (en la forma de una única capa bidireccional), tiene más pesos y es más complicado entrenarla. De hecho, vemos como en la primera época obtenemos valores de precisión tanto en validación como en entrenamiento inferiores a los visto nunca antes. Esto se alinea con nuestra hipótesis del infra-entrenamiento.

3.2 3.2. Modelo "más profundo"

En este apartado, vamos a partir nuevamente del modelo *Model_1* y vamos a añadir una segunda capa LSTM tras la primera, con el mismo tamaño y configuración, definiendo un nuevo modelo *Model_2*.

Entrénelo y compare los resultados.

[42]:
```
class Model_2(nn.Module):
  def __init__(self, feat_dim=20):
    super(Model_2, self).__init__()
```

```
    self.lstm1 = nn.LSTM(feat_dim, 256, batch_first=True, bidirectional=False)
    # Add a second LSTM layer, chaning its input size
    self.lstm2 = nn.LSTM(256, 256, batch_first=True, bidirectional=False)
    self.output = nn.Linear(256,1)

  def forward(self, x):
    # Forward the input through both LSTM layers
    out = self.lstm1(x)[0]
    out = self.lstm2(out)[0]
    out = self.output(out)
    out = torch.sigmoid(out)
    return out.squeeze(-1)
```

[43]:
```
# Define de model
model = Model_2(feat_dim=20)
model = model.to(torch.device("cuda"))

# Train and validate it
train_and_validate_model(model, model_name='2')
```

```
--- Model 2 ---
Epoch:  1
Loss: 0.11395259406052384
Train acc: 0.820111095905304
Val acc: 0.805751621723175
Epoch:  2
Loss: 0.07949710067580729
Train acc: 0.867836594581604
Val acc: 0.815490186214447
Epoch:  3
Loss: 0.06516276562915128
Train acc: 0.8848692774772644
Val acc: 0.9000653624534607
Epoch:  4
Loss: 0.061451548454808255
Train acc: 0.8878692984580994
Val acc: 0.8947712182998657
```

PREGUNTAS: 1. Incluya el código de la clase *Model_2* en la memoria.

Se ha incluido arriba. La única modificación relevante ha sido añadir la nueva capa al modelo, que tiene como entrada 256 conexiones (la salida de la capa LSTM anterior).

2. ¿Qué modelo obtiene un mejor resultado sobre los datos de validación, *Model_1* o *Model_2*? ¿Por qué puede ocurrir esto?

	Acc	Val Acc	Loss
1	0.8201	0.8058	0.1140
2	0.8678	0.8155	0.0795
3	0.8849	0.9001	0.0652
4	0.8879	0.8948	0.0615

Ambos modelos obtienen valores muy parecidos, tanto en entrenamiento como en validación. Finalmente tras la cuarta época el modelo 2 es ligeramente inferior al modelo 1. Aunque una capa adicional podría suponer una mejora en el modelo, se ha demostrado empíricamente que no por añadir más capas a una red obtendremos mejores resultados.

Sin embargo, la teoría más sencilla de por qué este modelo no es mejor que el modelo 1 es la misma que en el caso anterior: no hemos entrenado suficiente el nuevo modelo. Para obtener resultados significativos tendríamos que entrenarlo con un número mayor de épocas.

3. Y con respecto a *Model_1B*, ¿cuál es mejor?

De nuevo el modelo 2 es inferior a su opuesto. A pesar de que podríamos entrenar más ambos modelos para realizar otra comparativa, también es relevante admitir que tras poco entrenamiento y con el mismo número de neuronas (aunque con distinta topología de interconexión) el modelo 1B se entrena mejor con un número reducido de épocas. Es decir, una capa bidireccional obtiene mejores resultados con poco entrenamiento que dos capas LSTMs hacia delante.