PIT-Práctica 3

May 12, 2021

Diarización de Locutores mediante Redes Neuronales "End-to-End"

Antonio Coín Castro

12 de mayo de 2021

1 Implementando el sistema

1.1 Función de coste: BCE con permutational invariant training (PIT)

Para comenzar a implementar el sistema de diarización "end-to-end", vamos a empezar por la función de coste.

Tal y como se define en el artículo de referencia (sin tener en cuenta la parte de *deep_clustering*), la función de coste está definida como:

PREGUNTA:

- Con la ayuda del artículo de referencia, implemente la función *PITLoss* a continuación, e incluya el código en la memoria de la práctica.

Hay que tener cuidado porque la función `permutations` de `itertools` permuta en el primer eje, y tal y como se ha estructurado el código, lo que queremos es permutar en el último eje (el de los *speakers*).

[55]:
```
import torch
from itertools import permutations
from torch.nn.functional import binary_cross_entropy


def PITLoss(pred, target):
    """Calculate PIT loss.

    Input shape is (N, T, C), where:
        - N: batch size
        - T: sequence length
        - C: number of speakers

    Output is the average PIT loss for the whole batch. Note that
    PytTorch's built-in `binary_cross_entropy` already computes the
    average accross all axes, so there is no need to normalize.
```

```
    """
    min_loss = np.inf
    # Permute in speakers axis (2)
    for target_perm in permutations(target.swapaxes(0, 2)):
        # Reconstruct matrix from permuted vectors
        target_perm = torch.stack(target_perm)
        target_perm = target_perm.swapaxes(0, 2)  # Recover original shape
        loss = binary_cross_entropy(pred, target_perm, reduction='mean')
        if loss < min_loss:
            min_loss = loss
    return min_loss
```

Probamos manualmente que funciona bien en un ejemplo muy sencillo.

[ ]:
```
# Ground truth labels
target = np.array([
    # First batch element (seq1)
    [[0, 0, 1, 1, 1, 0, 0, 0],
     [0, 0, 0, 0, 0, 0, 1, 1]],
    # Second batch element (seq2)
    [[1, 0, 1, 1, 1, 0, 0, 0],
     [1, 0, 0, 0, 0, 0, 1, 1]],
], dtype=np.float64)

# Predicted labels (should actually be probabilities!)
pred = np.array([
    # First batch element (seq1)
    [[0.5, 0, 1, 1, 1, 0, 0, 0],  # <-- error in first component
     [0, 0, 0, 0, 0, 0, 1, 1]],
    # Second batch element (seq2)
    [[1, 0, 1, 1, 1, 0, 0, 0],
     [1, 0, 0, 0, 0, 0, 1, 1]],
], dtype=np.float64)

# Swap the last two dimensions and convert to tensors
target = torch.tensor(np.transpose(target, (0, 2, 1)))
pred = torch.tensor(np.transpose(pred, (0, 2, 1)))

bce = PITLoss(pred, target)
print(f"Average batch loss: {bce:.4f}")
```

```
Average batch loss: 0.0217
```

1.2 Definición del modelo

Ahora, vamos a definir el modelo de acuerdo al artículo:

PREGUNTA:

- Defina la clase BLSTM_5layers con la configuración del modelo del artículo: El modelo tiene 5 capas BLSTM de 256 unidades cada una, y recibe características de dimensión 23.

[56]:
```
import torch
import torch.nn as nn


class BLSTM_5layers(nn.Module):
    def __init__(self, feat_dim=23, nclasses=2):
        super(BLSTM_5layers, self).__init__()
        self.blstm = nn.LSTM(feat_dim, 256, batch_first=True,
                             bidirectional=True, num_layers=5)
        self.linear = nn.Linear(2*256, nclasses)

    def forward(self, x):
        """Input is (batch_size, seq_length, feat_dim)"""
        out = self.blstm(x)[0]
        out = self.linear(out)
        out = torch.sigmoid(out)
        return out
```

1.3 Entrenamiento del sistema

Ahora vamos a completar la implementación del sistema, para realizar su entrenamiento.

Reutilizando código de prácticas anteriores, recuerda los pasos principales, y ten en cuenta las peculiaridades para esta práctica:

- Cargar la lista de identificadores de entrenamiento.
- Declarar el modelo y cargarlo a la GPU.
- Declarar la función de coste (implementada previamente, PITLoss) y el optimizador.
- En lugar de generar minibatches con una longitud fija de secuencias, puedes utilizar un tamaño de batch de *1* y utilizar fichero a fichero.
- Para leer un audio puedes utilizar la función *read_recording* de prácticas anteriores.
- El tipo de parámetros o características (features) no tiene por qué ser exactamente el del artículo de referencia. Puedes utilizar por ejemplo *melspectrogram* de la librería *librosa*.
- Puesto que las secuencias de características y las etiquetas deben tener la misma longitud, y el RTTM puede contener silencio al final, puedes realizar un *padding* de las etiquetas a 0 hasta completar la longitud del audio (o recortar la secuencia de *features*).

PREGUNTA:

- Completa el código siguiente e inclúyelo en la memoria.

Para calcular el melespectrograma, tenemos en cuenta que el sampling rate es $fs = 16000$, por lo que para implementar las medidas de referencia del artículo de ventanas de 25ms y saltos de 10ms, escogemos

$$hop = \frac{16000}{1000} \cdot 10 = 160$$

$$win = \frac{16000}{1000} \cdot 25 = 400$$

Además, damos la opción tanto de hacer *padding* con ceros a las etiquetas como de recortar la longitud de las *features* (hay que escoger una). Nosotros elegimos para entrenar el modelo la segunda opción, y entrenamos durante 5 épocas con optimizador Adam (*learning rate* $= 10^{-3}$).

[8]:
```
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
```

[57]:
```
from torch import optim
import librosa

# LISTA DE ENTRENAMIENTO
TRAIN_PATH_FEATS = 'data/train_2spk/wav/'
TRAIN_PATH_LABS = 'data/train_2spk/rttm/'
with open('train_DIAR_2spk.lst', 'r') as f:
    train_list = f.read().splitlines()

# MODELO
device = torch.device("cuda")
model = BLSTM_5layers()
model = model.to(device)

# CRITERIOS DE OPTIMIZACIÓN
criterion = PITLoss
optimizer = optim.Adam(model.parameters(), lr=0.001)

# PARÁMETROS DE ENTRENAMIENTO
trim_features = True
max_iters = 5

# ENTRENAMIENTO
for epoch in range(1, max_iters + 1):
    model.train()
    cache_loss = 0.0

    # Batch size is 1
    for file_id in train_list:
        optimizer.zero_grad()

        # Create training sample
```

```
        fs, signal = read_recording(TRAIN_PATH_FEATS + file_id + ".wav")
        # 25ms frame length and 10ms frame shift (at fs=16000)
        feats = librosa.feature.melspectrogram(
            signal, fs, n_mels=23, win_length=400, hop_length=160)
        feats = feats.T  # n_frames x n_features
        labs = rttm2mat(TRAIN_PATH_LABS + file_id +
                        ".rttm")  # n_frames x n_speakers
        if trim_features:
            feats = feats[:len(labs), :]
        else:
            labs = np.pad(labs, [(0, len(feats) - len(labs)), (0, 0)])

        # Make sure that all frames have defined label
        assert len(labs) == len(feats)

        # Place data into Pytorch tensors
        feats = torch.tensor(
            feats[np.newaxis, :].astype("float32")).to(device)
        labs = torch.tensor(
            labs[np.newaxis, :].astype("float32")).to(device)

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

        # Compute cost
        loss = criterion(outputs, labs)

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

    epoch_loss = cache_loss/len(train_list)
    print(f'[{epoch:02d}] loss: {epoch_loss:03f}')
```

```
[01] loss: 0.611003
[02] loss: 0.580144
[03] loss: 0.578037
[04] loss: 0.579590
[05] loss: 0.575558
```

1.4 Evaluación: DER (Diarization Error Rate)

Finalmente, realiza la evaluación de dos ficheros al azar (dentro del conjunto de entrenamiento) y obtén el DER para cada uno de ellos utilizando el toolbox *dscore* presentado al inicio de la práctica.

PREGUNTA:

- Incluye en la memoria el código utilizado, los identificadores de los ficheros elegidos y sus

correspondientes errores (DER).

[59]:
```
import librosa


def predict(file_id):
    model.eval()
    with torch.no_grad():
        fs, signal = read_recording(TRAIN_PATH_FEATS + file_id + ".wav")
        feats = librosa.feature.melspectrogram(
            signal, fs, n_mels=23, win_length=400, hop_length=160)
        feats = torch.tensor(
            feats.T[np.newaxis, :].astype("float32")).to(device)
        outputs = model(feats).cpu()[0]
    return outputs
```

Elegimos los ficheros con índices 5 y 10, correspondiente a los identificadores `djngn` y `gpjne`.

[60]:
```
# Create rttm for predictions
valid_idx = [5, 10]
for idx in valid_idx:
    file_id = train_list[idx]
    preds = predict(file_id)
    mat2rttm(preds, file_id, file_id + '_pred.rttm')
```

[61]:
```
!python dscore/score.py - r data/train_2spk/rttm/djngn.rttm \
    data/train_2spk/rttm/gpjne.rttm \
    - s djngn_pred.rttm gpjne_pred.rttm
```

```
Loading speaker turns from reference RTTMs…
Loading speaker turns from system RTTMs…
WARNING: No universal evaluation map specified. Approximating from reference and
speaker turn extents…
Trimming reference speaker turns to UEM scoring regions…
Trimming system speaker turns to UEM scoring regions…
Checking for overlapping reference speaker turns…
Checking for overlapping system speaker turns…
Scoring…
File               DER    JER    B3-Precision    B3-Recall    B3-F1    GKT(ref,
sys)    GKT(sys, ref)    H(ref|sys)    H(sys|ref)    MI    NMI
---------------  -----  -----  --------------  -----------  -------
---------------  ---------------  ------------  ------------  ----  -----
djngn            33.97  66.17           0.54          1.00     0.70
1.00             0.00          1.11          0.00  0.00   0.00
gpjne            24.83  61.53           0.61          1.00     0.76
1.00             0.00          1.00          0.00  0.00   0.00
*** OVERALL ***  29.15  63.85           0.57          1.00     0.73
1.00             0.40          1.05          0.00  1.00   0.70
```

Como vemos, obtenemos un DER de 33.97% para el archivo `djngn` y un DER de 24.83% para `gpjne`. Con los dos archivos elegidos, la red predice (incorrectamente) que durante toda la longitud del audio está hablando uno de los dos *speakers*, por lo que solo acierta de casualidad en aquellos segmentos donde de verdad esté hablando ese speaker.

Hemos intentado también la otra opción de hacer padding con ceros en la parte final de las etiquetas. En este caso, lo que se observaba es que la red aprendía a predecir que había silencio durante toda la longitud del audio, por lo que los ficheros *rttm* resultantes estaban vacíos, y se obtenía un DER del 100% (había fallos de *missed speech* en toda la longitud del habla).

La conclusión que sacamos es que la red no ha conseguido aprender con los pocos ejemplos de los que disponemos, y que necesitaríamos bastantes más para empezar a ver buenos resultados. Notamos que todas las predicciones se realizan con un umbral de 0.5 tras la salida de la sigmoide, y probablemente si reducimos un poco este umbral cambien los comportamientos de las predicciones que acabamos de mencionar.

PREGUNTA:

- ¿Cómo se interpreta el DER?

El *Diarization Error Rate* puede interpretarse como la proporción del tiempo total de habla donde hay fallos de algún tipo, teniendo en cuenta posibles solapamientos. Es algo así como un *accuracy* global ponderado en el tiempo, y es mejor cuanto más bajo sea, siendo 0 el valor asociado a una predicción perfecta.

Si nos referimos al etiquetado original como "referencia" y al etiquetado predicho por nuestra red como "sistema", podemos expresar el DER como

$$DER = \frac{MS + FA + SPE}{TOTAL} \cdot 100\%,$$

donde:

- MS (*Missed Speech*) es el tiempo total de referencia que no ha sido atribuido a ningún speaker en el sistema.
- FA (*False Alarm*) es el tiempo total del sistema que no es asignado a un speaker de referencia.
- SPE (*Speaker Error*) es el tiempo total de referencia atribuido a un speaker equivocado en el sistema.
- TOTAL es el tiempo total de habla de referencia (ignorando silencios).

En el caso de solapamiento, los valores se ven ponderados por el número de speakers implicados, tal y como se describe en el Rich Transcription Meeting Recognition Evaluation Plan 2009 del NIST (sección 6.1).

- ¿Puede tener un valor mayor de 100%?

Sí, el DER no está acotado superiormente. Por ejemplo, podemos considerar un audio donde un speaker habla durante el primer 20% del tiempo, el 60% siguiente es silencio, y otro speaker habla durante el 20% final. Si nuestro sistema predice que el primer speaker habla durante toda la longitud del audio, digamos N, obtendríamos un DER de

$$DER = \frac{0.0N\ (MS) + 0.6N\ (FA) + 0.2N\ (SPE)}{0.4N\ (TOTAL)} \cdot 100\% = 200\%.$$

Vemos que aquí superamos el 100% porque el denominador no tiene en cuenta la longitud total del audio, sino solo la del habla.