Implementation of SELD-TCN: Sound Event Localization and Detection via Temporal Convolutional Network

Giuseppe Sensolini Arrà

Sapienza università di Roma
Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Antonio Ruberti

August 2020

1 Introduction

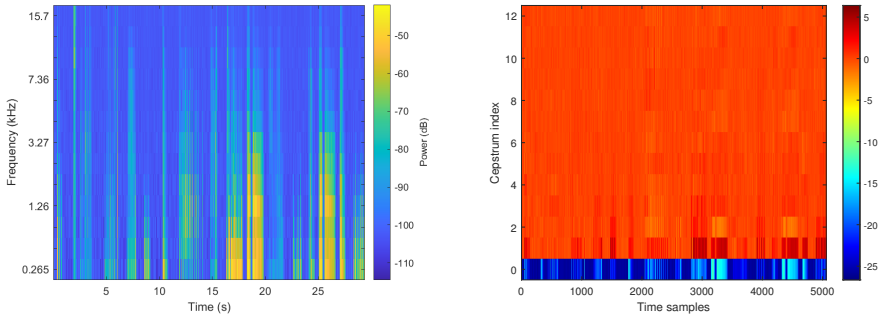
In this technical report the implementation of "*SELD-TCN: sound event localization & detection via temporal convolutional networks*" will be discussed. SELD-TCN address two separate problems: *sound event detection* (SED) and *direction of arrival* (DOA) estimation.

The aforementioned implementation can be found in [5].

The neural network has been developed in Linux environment using Python 3.8, Tensorflow 1.14.0 and Keras 2.2.4. Preprocessing and evaluation metrics have been largely taken from [4].

2 Implementation

Dataset Preprocessing. The main dataset used is part of [6], containing 360 multichannel audio recordings and 11 sound categories, each 30 seconds long and sampled at 44.1 kHz. Short Fourier transform (STFT) was used in order to obtain the spectrograms (with an hamming windows of length 512 samples and 50% overlap). Inputs for the first layer of the network are represented by both phase and magnitude of the spectrograms.
An example of preprocessing done in Matlab are shown in *Figure 1*. The same operation is repeated for each channel (4 with this dataset), and for each audio track.

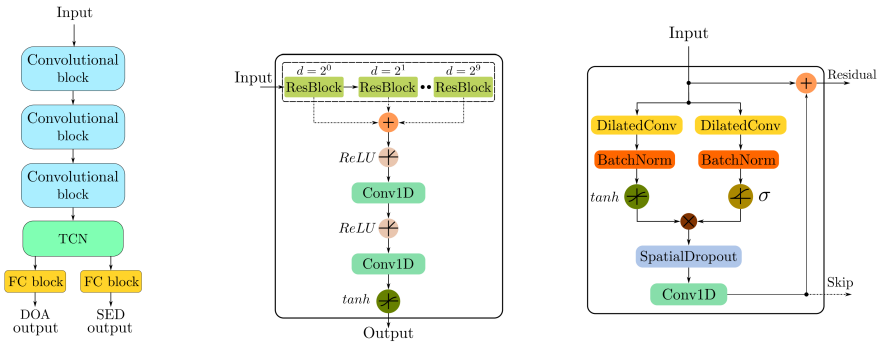


(a) Spectrogram (b) Mel spectrogram

Figure 1: pre-processing examples

SELD-TCN. In this paragraph the implementation of the network will be discussed. The main innovation brought by SELD-TCN with respect to SELDnet [4] is the introduction of a *Temporal convolutional block*. The TCN block is the core of the entire architecture; it utilize dilated convolutions to enlarge the receptive field, allowing a wider part of the input data to contribute to the output. Clearly, receptive field size can be easily changed by changing the dilatation rate or kernel sizes.
Some important motivations behind the adoption of TCNs are summarized here:

- TCNs process the whole audio sequence in parallel; this guarantees an important speed-up with respect to a classical sequential processing.
- The architecture can take a sequence of any length and map it to an output sequence of the same length, just as with an RNNs.
- The convolutions in the architecture are causal, meaning that there is no information dispersion from future to past.
- Low memory requirement for training. GRUs (used is SELDnet) can easily use up a lot of memory to store the partial results, while in a TCN the filters are shared across layers.



(a) SELD-TCN architecture (b) TCN block architecture (c) ResBlock architecture

Figure 2: An overview of the SELD-TCN architecture

Neural network starts with three consecutive *convolutional blocks* (*Figure 2*), designed to learn both intra-/ and inter-channel features. Each convolutional block is composed by:

1. 2D convolutional layer (64 filters of size 3x3)
2. Batch normalization layer
3. ReLU activation layer

4. 2D max pooling (1x8 in 1^{st} and 2^{st} convolutional blocks, 1x2 in the 3^{st})

These blocks are implemented as follows:

```
for i, convCnt in enumerate(pool_size):
        spec_cnn = Conv2D(filters=nb_cnn2d_filt, kernel_size=(3, 3),
                          padding='same')(spec_cnn)
        spec_cnn = BatchNormalization()(spec_cnn)
        spec_cnn = Activation('relu')(spec_cnn)
        spec_cnn = MaxPooling2D(pool_size=(1, pool_size[i]))(spec_cnn)
        spec_cnn = Dropout(dropout_rate)(spec_cnn)
spec_cnn = Permute((2, 1, 3))(spec_cnn)
```

The output of this first group of layers is used to feed the TCN block. It represents the core of the whole architecture. The idea of TCN block has been initially adopted in [2]. TCNs typically utilize causal convolutions, where the output at time t depends solely on the current and past elements; in this case a modified version (non-causal) is used. The basic idea here is to learn the underlying structure in a sequence by using dilated convolutions.

Inside the TCN block ten *Residual blocks* are used, each one with a different level of dilatation (from 2^0 up to 2^9). *Dilated convolutions* allow a wider part of the input data to contribute to the output, guaranteeing large receptive fields. Receptive field sizes can be changed through increasing their dilatation rates or kernel sizes. 256 filters of size 3 are used here.
After that two parallel activation layers are applied, respectively with tanh and sigmoid activations, and then the two parallel branches are mixed back together. A *spatial dropout* layer (with dropout rate of 0.5) and another 1D convolution (256 filters of size 1) follows. This operations are repeated ten times, one for each *ResBlock*, and the sum of the outputs is convolved two times again; the implementation follows:

```
skip_connections = []
resblock_input = Reshape((data_in[-2], -1))(spec_cnn)

for d in range(10):
        # 1D convolution + Batch normalization
        spec_conv1d = keras.layers.Convolution1D(filters=256,
                                                 kernel_size=(3),
                                                 padding='same',
                                                 dilation_rate=2**d)
                                                 (resblock_input)
        spec_conv1d = BatchNormalization()(spec_conv1d)

        # parallel activations
```

```
        tanh_out = keras.layers.Activation('tanh')(spec_conv1d)
        sigm_out = keras.layers.Activation('sigmoid')(spec_conv1d)
        spec_act = keras.layers.Multiply()([tanh_out, sigm_out])

        # spatial dropout
        spec_drop = keras.layers.SpatialDropout1D(rate=0.5)(spec_act)

        # 1D convolution
        skip_output = keras.layers.Convolution1D(filters=128,
                                                 kernel_size=(1),
                                                 padding='same')
                                                 (spec_drop)

        # Residual output
        res_output = keras.layers.Add()([resblock_input,skip_output])

        if skip_output is not None:
                skip_connections.append(skip_output)

        # prepare input for next ResBlock
        resblock_input = res_output

# Residual blocks sum + activation
spec_sum = keras.layers.Add()(skip_connections)
spec_sum = keras.layers.Activation('relu')(spec_sum)

# 1D convolution + activation
spec_conv1d_2 = keras.layers.Convolution1D(filters=128,
                                           kernel_size=(1),
                                           padding='same')
                                           (spec_sum)
spec_conv1d_2 = keras.layers.Activation('relu')(spec_conv1d_2)

# 1D convolution + activation
spec_tcn = keras.layers.Convolution1D(filters=128,
                                      kernel_size=(1),
                                      padding='same')
                                      (spec_conv1d_2)
spec_tcn = keras.layers.Activation('tanh')(spec_tcn)
```

Clearly, the total number of layers used here is larger with respect to the ones used in the *Recurrent block* proposed in [4].

A last important element of the network is the *Fully-connected block*. It contains 2 parallel branches, addressing both SED and DOA.
Sound event detection (SED) is performed by a dense layer with 128 neurons. Note also the presence of *TimeDistributed()*: it allows to apply a layer to every temporal slice of an input. It follows a dropout layer and a last dense layer, with sigmoid activation and number of neurons equal to 3xN_{SED} (i.e., the number of events of the dataset, 11 in this case). The output range is in [0,1], with 1 being for active sound event.

```
# Sound Event Detection (SED)
sed = spec_tcn
for nb_fnn_filt in fnn_size:
        sed = TimeDistributed(Dense(nb_fnn_filt))(sed)
        sed = Dropout(dropout_rate)(sed)
sed = TimeDistributed(Dense(data_out[0][-1]))(sed)
sed = Activation('sigmoid', name='sed_out')(sed)
```

A similar procedure is applied to the DOA branch; in this case the final dense layer has 3xN_{SED} neurons (each denoting the 3D cartesian coordinates at the origin of a unit sphere around the microphone array). The activation is of type tanh with output range in [-1,1] for x,y and z axis.

```
# Direction of Arrival (DOA)
doa = spec_tcn
for nb_fnn_filt in fnn_size:
        doa = TimeDistributed(Dense(nb_fnn_filt))(doa)
        doa = Dropout(dropout_rate)(doa)

doa = TimeDistributed(Dense(data_out[1][-1]))(doa)
doa = Activation('tanh', name='doa_out')(doa)
```

3 Results

1,469,097 trainable parameters are present, about three times more with respect to [3]; the discussed network achieves also faster training time per epoch. The SELD-TCN was trained with the Adam optimizer with default parameters and batch size of 16. Although the original paper utilizes 500 epochs, in this simulation 200 epochs seems to be enough to reason about the correctness of the above shown implementation.
The evaluation is mainly based on the armonic mean of the precision and recall (F1), error rate (ER) and DOA error (DE). The obtained results are consistent with the ones achieved in [1], they are summarized in *Table 1*, while F1 and ER improvements over time are shown respectively in *Figure 3* and *Figure 4*.

F1	**ER**	**DE**
94.6	0.091	17.1

Table 1: SELD-TCN results

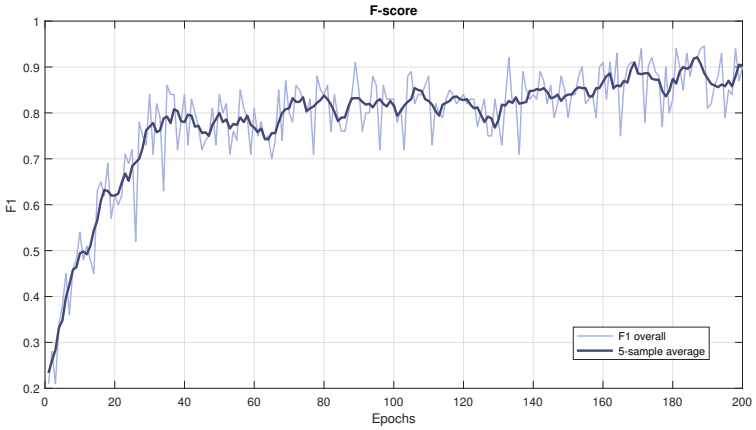


Figure 3: Harmonic mean of the precision and recall (F1)

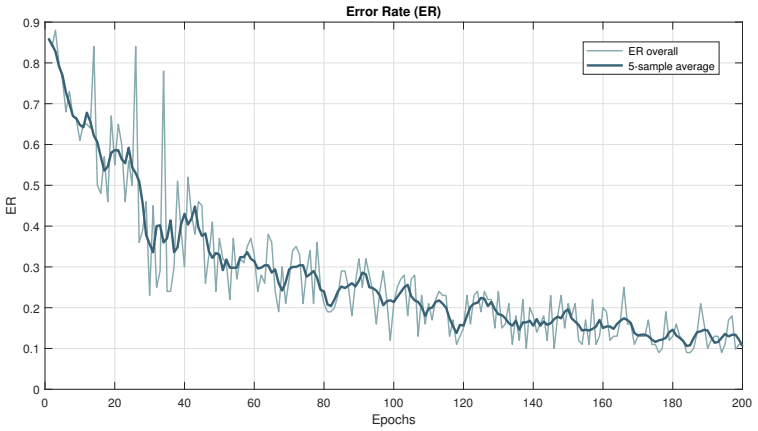


Figure 4: Error rate (ER) with regard to segment-wise substitutions, insertions and deletions

References

[1] K. Guirguis, C. Schorn, A. Guntoro, S. Abdulatif, B. Yang, *SELD-TCN: sound event localization & detection via temporal convolutional networks.*

[2] D. Rethage, J. Pons, X. Serra, *A Wavenet for Speech Denoising.*

[3] S. Adavanne, A. Politis, J. Nikunen, T. Virtanen, *Sound Event Localization and Detection of Overlapping Sources Using Convolutional Recurrent Neural Networks.*

[4] S. Adavanne, *SELDnet code:*
`https://github.com/sharathadavanne/seld-net.git`

[5] G. Sensolini Arrà, *SELD-TCN code:*
`https://github.com/giusenso/seld-tcn.git`

[6] ANSIM dataset, *TUT Sound Events 2018 - Ambisonic, Anechoic and Synthetic Impulse Response Dataset.*