NEURAL NETWORKS PROJECT

-

Light-SERNet: A lightweight fully convolutional neural network for speech emotion recognition

Gianluca Panici, 1666962

June 2022

ABSTRACT

This document is the report for the project done for the Neural Networks course at Sapienza University of Rome.

The project consists in studying, analyzing and testing a given academic research in order to understand and experiment the state-of-the-art in this branch of computer science.

The study in object is *"Light-SERNet: A lightweight fully convolutional neural network for speech emotion recognition"* [1]. It deals with the detection of the emotion of the speaker in a speech signal with an efficient and lightweight system, suitable for low power devices.

In my work I studied the paper, trained the model, compared the obtained result and then performed some inference tests. While the training outcomes were consistent with the paper's described results, my personal tests deviated from my expectation, and an explanation is given for this.

To complete my project I forked the original repository, adding some adjustments for my purposes. The code can be viewed at `https://github.com/PanK0/LIGHT-SERNET`.

In the dedicated section [1.1] of this report I will refer to the code provided in [3], starting from the file *train.py* [4] for discovering all the most significant parts of the project. Then the training of the model is analyzed and some tests are discussed.

1 Light-SERNet

Light-SERNet is a lightweight fully convolutional neural network for speech emotion recognition.

With the evolution of the technological scenario where the human-computer interaction has become a daily constant in our society, the detection of human emotions from a speech signal plays an important role in this exchange between humans and machines. [1]

Existing benchmarks of speech emotion recognition (**SER**) methods are mainly comprised of a feature extractor and a classifier to obtain the emotional states. Recently, deep learning techniques are used to extract high level features from raw data and it is shown that they are effective for speech emotion recognition [5]

In this study, convolutional neural networks (**CNNs**) are used, due to the significant improvements in SER. This kind of tools are useful especially when it is required to divide the information in smaller chunks to analyze them and find out some characteristics that may appear irrelevant to the target task, and so they find their perfect place when the input is a complex unstructured signal, such as image or a speech signal. [1]

1.1 Architecture

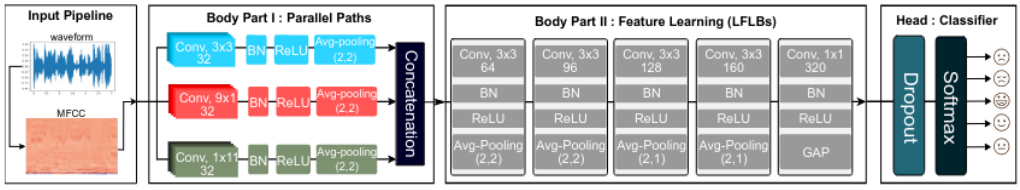


Figure 1: High level architecture of Light-SERNet

The architecture is composed of **three main parts**:

- input pipeline
- body
- classification block

The model is defined into the file named *models.py* [6] given at [3].

```
def Light_SERNet_V1(output_class, input_duration,
    input_type="mfcc"):
```

1.1.1 Input pipeline

After normalizing the audio signals between -1 and 1, the Mel Frequency Cepstral coefficients (**MFCCs**) of the signals are calculated. To this end, it is used a Hamming window to split the audio signal into 64-ms frames with 16ms overlaps, which can be considered as quasi-stationary segments. Following a 1024-point Fast Fourier Transform (**FFT**) applied to each frame, the signal undergoes a Mel scale filter bank analysis, in the range of 40Hz to 7600Hz. The MFCCs of each frame are then calculated using and inverse discrete cosine transform, where the first 40 coefficients are selected to train the model. [1]

In the code The first action is to **segment** the dataset. In the code provided at [3], this job is achieved by running the script contained in *segment_dataset.py* [7], called from *train.py* [4].

The segmentation consists in treating the input in order to obtain **same sized** and **normalized** audio files, i.e. files with a same fixed length and with the same level for the entire duration.

Then, in the file *train.py* [4] the audio preprocessing is performed in the function `make_dataset_with_cache()`, defined in file *dataio.py* [8]. This function, other than generating a volatile instance of the dataset to reduce the latency induced by accessing the disk, also takes care of the **input preprocessing** through the function `preprocess_dataset()`.

At this point, depending of which indications we gave in input for the training, we can train the neural network by processing the segmented files to obtain a **spectrogram**, a **mel spectrogram** or a **MFCC** (default) representation of the audio signal. (Figure 2)

This work is done by a series of subroutines inside *dataio.py* [8] called by the function `get_input_and_label_id()` that is invoked in `preprocess_dataset()`.

The resulting **output** from `make_dataset_with_cache()` is the dataset splited into `train` and `test` sets, with the files properly processed to correctly feed the neural network.

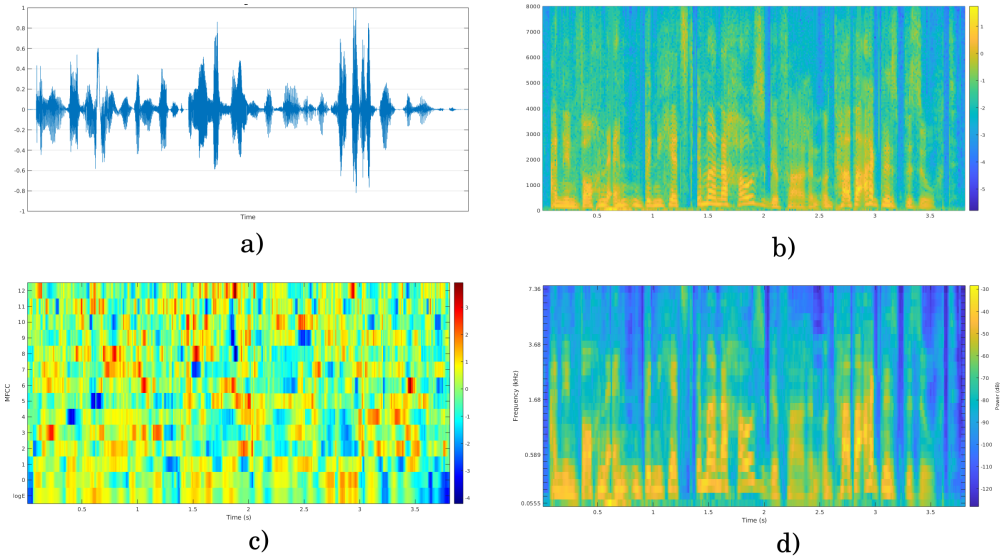


Figure 2: Different representations of the same audio signal: **a)** waveform, **b)** spectrogram, **c)** MFCC, **d)** mel spectrogram

1.1.2 Body Part I

Some components comes in the game when dealing with increasing the classification accuracy: the receptive field size, the network depth (i.e. the number of layers), and width (i.e. the number of filters per layer) and batch normalization just to give an example. [9]

In **Body Part I** (Second block of Figure 2) the information is routed in **three parallel paths**, in which for each dimension of the multi-dimensional signal a receptive field is calculated. Hence, kernels of size 9×1, 1×11 and 3×3 to extract spectral, temporal and spectral-temporal dependencies, respectively, are used (Figure 3)

The advantage of using this technique over having only one path with the same receptive field size is to **reduce the number of parameters and the computational cost** of this part of the model. [1]

At the end, the extracted features of each path are concatenated and fed into Body Part II. [1]

In the code In the code provided at [3], this part is built into the module named *models.py* [6], lines $34 - 53$.

```
...

path1 = layers.Conv2D(32, (11,1), padding="same",
    strides=(1,1))(body_input)
path2 = layers.Conv2D(32, (1, 9), padding="same",
    strides=(1,1))(body_input)
path3 = layers.Conv2D(32, (3, 3), padding="same",
    strides=(1,1))(body_input)

path1 = layers.BatchNormalization()(path1)
path2 = layers.BatchNormalization()(path2)
path3 = layers.BatchNormalization()(path3)

path1 = layers.ReLU()(path1)
path2 = layers.ReLU()(path2)
path3 = layers.ReLU()(path3)

path1 = layers.AveragePooling2D(pool_size=2, padding="same")(path1)
path2 = layers.AveragePooling2D(pool_size=2, padding="same")(path2)
path3 = layers.AveragePooling2D(pool_size=2, padding="same")(path3)


feature_extractor = tf.keras.layers.Concatenate(axis=-1)([path1,
    path2, path3])

...
```

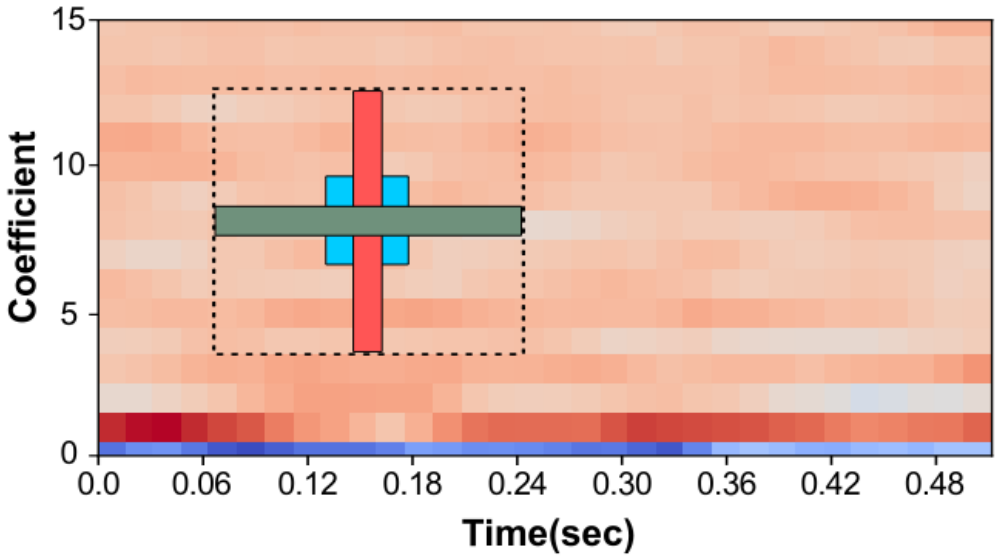


Figure 3: The effect of parallel paths (gray, red and blue rectangles) and their resulting receptive fields (dotted line rectangle)

1.1.3 Body Part II

The **Body Part II** consists of several LFLBs with different configurations. An **LFLB** is a collection of **consecutive layers**. The original LFLB consists of a **convolution layer**, a **batch normalization layer (BN)**, an **exponential linear unit (ELU)** and a **max-pooling layer**. Here, the ELU layer and the max-pooling layer have been replaced by a **rectified linear unit (ReLU)** and the **average pooling**, respectively. [1] This configuration is shown in Figure 2, in the third block.

In the code In the code consultable at [3], this setup is done in the module named *models.py* [6], lines $55 - 79$.

1.1.4 Head

The body part is supposed to map the nonlinear input space into a linearly separable sub-space, so one fully-connected layer is enough for the classification. Therefore, the head part only includes a **dropout layer** to reduce overfitting and a **fully-connected layer** with a softmax activation function that reduces the computational complexity and the number of parameters. [1]

In the code In the code given at [3], this setup is done in the module named *models.py* [6], lines $82 - 85$.

```
...

x = layers.Dropout(hyperparameters.DROPOUT)(x)
body_output = layers.Dense(output_class, activation="softmax")(x)

...
```

1.2 Dataset

In the study [1] two datasets have been used: **IEMOCAP** [10] and **EMO-DB** [11], while for this project only EMO-DB has been used.

EMO-DB EMO-DB is a German-language dataset, recorded by ten professional actors and actresses (five men and five women). The dataset includes 535 emotional utterances in 7 classes: anger (23.7%), natural (14.7%), sadness (11.5%), fear (12.9%), disgust (8.6%), happiness (13.2%) and boredom (15.1%). [1]

2 Training and Results Comparison

In this section I am going to present the **steps for training** the Neural Network, from the input to the output, directly **referring to the code** in the file *train.py* [4]

Experimental Setup For this work has been used a similar setup to the one used in the main study, with the major difference residing in the hardware: for this project, the models have been trained using Google Colab with GPU hardware accelerator.

In the file *hyperparameters.py* [12] all the hyperparameters for the training are set: models are trained for 300 epochs with a batch size of 32, using an Adam optimizer.

To reduce **overfitting** a batch normalization is used after each convolutional layer, with a dropout at a rate of 0.3 before the softmax layer.

2.1 Training

The magic happens in the file *train.py* [4].

To start, **clone the repository** given at [3], **move into it** and **install the requirements**.

```
$ pip install -r requirements.txt
```

As said before, my experiments have been done with the EMO-DB dataset [11], so it is needed to **download it** and to **unzip it into the folder *data*** of the repository.

Input To start the training, some information has to be given in the command line:

```
$ python train.py  -dn {dataset_name}
                   -id {input durations}
                   -at {audio_type}
                   -ln {cost function name}
                   -v {verbose for training bar}
                   -it {type of input(mfcc, spectrogram,
                      mel_spectrogram)}
```

Where:

- `-dn {dataset_name}` : name of the dataset, like `EMO-DB` or `IEMOCAP`.
- `-id {input_durations}` : desired duration of the audio files, that can be set to 3 for booth EMO-DB and IEMOCAP or to 7 for IEMOCAP only.
- `-at {audio_type}` : can be set to `all` (for both EMO-DB and IEMOCAP) and, for IEMOCAP only, to `impro`, for improvised part, ot to `script`, for scripted part.
- `-ln {cost function name}` : loss function name. Here `focal` (for focal loss) or `cross_entropy` functions can be used.
- `-v {verbose for training bar}` : accepts values 0 or 1.
- `-it {type of input}` : for analyze the audio files in `mfcc`, `spectrogram` or `mel_spectrogram`.

So, for the training of the model I am analyzing in this work I used the following input:

```
$ python train.py  -dn "EMO-DB"
                   -id 3
                   -at "all"
                   -ln "cross_entropy"
                   -v 1
                   -it "mfcc"
```

Dataset Segmentation As specified in 1.1.1, the dataset need to be segmented to transform the input in some data that can be properly processed by the Neural Network. In file *train.py* [4] this is done in lines $81 - 89$.

Start the effective training In file *train.py* [4] the effective training is performed in lines $109 - 197$.

To train the model a **K-FOLD** Cross Validation is performed with $K = 10$. So, for each folding a test set and a training set are created by the function `make_dataset_with_cache()`, defined in file *dataio.py* [8], and a model is generated, trained and tested with those data.

During the training and testing phases in the various foldings of the K-FOLD the **weights** of the model that performed the best results in terms of **accuracy** are saved.

At the end of the K-FOLD, the **best weights are loaded into three models** with different degrees of precisions in terms of information saved, such as `Float32`, `Float16` and `int8`. This is done in file *train.py* [4] at lines $202 - 240$. Three models with different sizes are saved to stay in line with the purpose of the main study, i.e. to obtain a lightweight model that can be **utilized in system with limited hardware**.

The models are finally **evaluated** by the function `evaluate_model()`, that calls the `run_tflite_model()` function in help. Both are defined in file *tflite_evaluate.py* [13]

2.2 Comparison of obtained results

For the study two loss functions are used to train the proposed models: **Focal loss** (F-Loss) and **cross-entropy loss** (CE-Loss). In the experiments, F-Loss with $\gamma = 2$ is used.

The **metrics** used for the performances evaluation of the proposed models are: 1) unweighted accuracy (**UA**), 2) weighted accuracy (**WA**) and 3) F1-score (**F1)**).

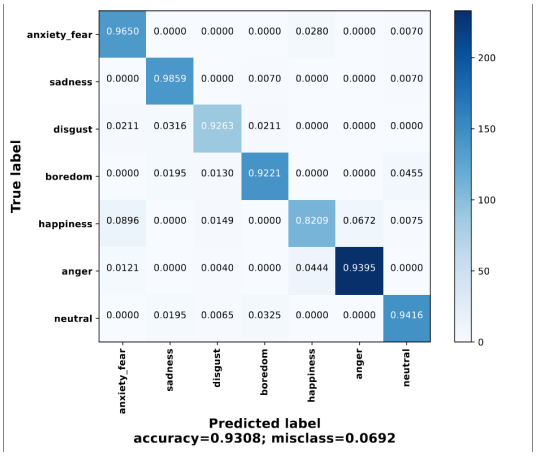
The results obtained in the study suggest that for EMO-DB dataset the Cross Entropy Loss function performs better. These results indicate that the UA of the models can improve the performance, in some cases, with simple CE-Loss. [1]

Consistent results came out from my experiments: as shown in **Table 1**, I obtained similar values for accuracy and F1-score with the only difference in the UA being now higher with the F-Loss instead of while using CE-Loss, but the gap is so minimal that can be negligible.

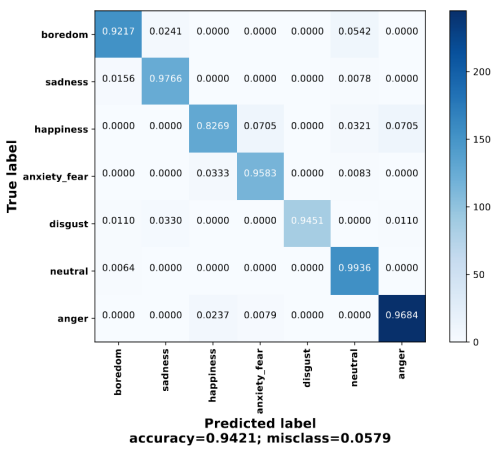
	EMO-DB					
	F-Loss			**CE Loss**		
	UA	**WA**	**F1**	**UA**	**WA**	**F1**
Study results	92.88	93.08	93.05	94.15	94.21	94.16
My results	94.94	95.14	95.10	94.89	95.33	95.27

Table 1: The performance models for EMO-DB dataset with input lengths of 3 seconds in terms of UA(%), WA(%) and F1(%) obtained in the study and in my experiments

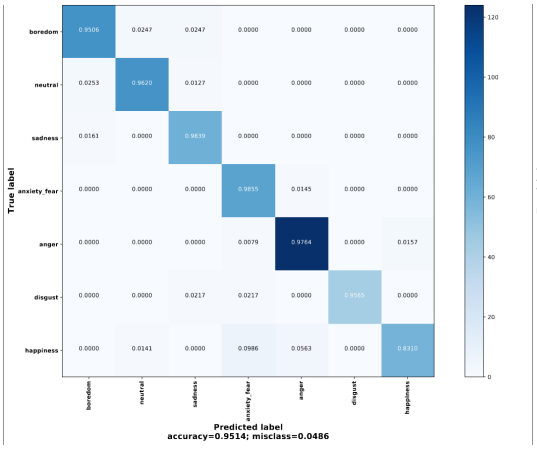
In Figure 4 I also reported the confusion matrices obtained by my experiments (Fig. 4c and Fig. 4d), comparing them with those provided in the study (Fig. 4a and Fig. 4b).



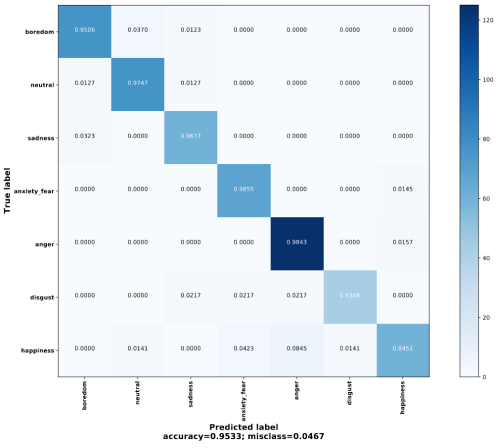
(a) Study result - Focal Loss



(b) Study result - Cross Entropy



(c) My result - Focal Loss



(d) My result - Cross Entropy

Figure 4: Confusion matrices provided in the study (Fig. 4a and Fig. 4b) and obtained by my experiments (Fig. 4c and Fig. 4d), using both Focal Locc and Cross Entropy Loss functions.

3 Inference Tests

To do some inference tests I created a dedicated directory named *inference_tests* [14] in which I put some useful material to play a little with.

In this directory it is possible to find the a *.tflite* trained model with cross entropy in Float32 precision with the dedicated reports and some audio files that can be used to test the model.

Among the audio files some belong to the EMO-DB dataset and some are personally recorded by me. To distinguish the files, I used some simple rules to name them, such that each audio file contains some information in

its own name:

- The **belonging class** label
- If the file **belongs to the EMO-DB Dataset** it is named with `dataset` label
- If the file is a **phrase from the dataset** but it has been **recorded by me** from a german speaker it is named with `rec` label
- If the file is **NOT a phrase from the dataset** and it has been **recorded by me** from a german speaker it is named with `external` label

However, when executing some tests with the files recorded by me, there is something important to consider: my audio files are recorded using my own smartphone with two performers of which just one was a mother-tongue German speaker and none of them is a professional actress.

In my inference tests all the files belonging to the dataset were correctly classified, while on 7 files recorded by me I noticed some inconsistencies.

As shown in Table 2 some files were misclassified. This may be due to different causes that can be the fact that the speakers were not professional actresses, some background noise or my devices's voice recorder that is not a professional one. Each one of these elements could add features to the final audio that, when preprocessed, may completely change the classification.

	File Name	Origin	Correct Class	Classified as
1	sadness_external.wav	external	SADNESS	BOREDOM
2	sadness_external_2.wav	external	SADNESS	BOREDOM
3	happiness_external.wav	external	HAPPINESS	HAPPINESS
4	anger_external.wav	external	ANGER	ANGER
5	disgust_rec.wav	recorded	DISGUST	DISGUST
6	boredom_rec.wav	recorded	BOREDOM	BOREDOM
7	anger_rec.wav	recorded	ANGER	HAPPINESS

Table 2: Classification of the files recorded by me. Those marked with **external origin** are phrases that DO NOT belong to the dataset, while those marked with **recorded** are recorded audios of dataset phrases.

In the code In the code provided at [3] I execute my inference tests by running the script contained in the file *inference_single_file.py* [15] that, accepting as input the audio duration, audio type, loss function, input type and input file name, returns the classification of the given file.
Here is an example of input for calling this simple program:

```
$ python inference_single_file.py -id 3 -at "all" -ln
    "cross_entropy" -it "mfcc" -fn "happiness_dataset.wav"
```

To do the trick, this file calls the help of some functions defined in *inference_data_processing.py* [16], that basically let the input audio file do the same path of the input pipeline to obtain some data edible for the classifier.

First of all, the file is **segmented** and **normalized** (default segment length = 3), then it is **preprocessed** to obtain the input type we want (default: **mfcc**, other possible: spectrogram, mel-spectrogram) and finally the **model is loaded** so that it can anlyze the preprocessed input and return as output the belonging class of the audio file.

The following code shows the three steps presented above in the file *inference_single_file.py* [15].

```
...

# ***** SEGMENTING DATA *****
segmented_file = segment_file(filename, input_duration,
    segment_mode=1)

# ***** INPUT PREPROCESSING *****
preprocessed_input = preprocess_input(segmented_file, input_type)

# ***** LOAD AND RUN THE MODEL *****
model_path =
    f"inference_tests/EMO-DB_3.0s_Segmented_cross_entropy_float32.tflite"
predictions = run_model(model_path, preprocessed_input)

...
```

Conclusions

As described in the original study, being able to make speech emotion recognition applications portable could be an important improvement for the pervasiveness of technology in the next future.

More and more applications nowadays take advantage of vocal input commands given by the user and to distinguish the emotion of the speaker can completely change the response of the vocal assistant.

Just to give a short example to scratch the surface of this ability, we can consider a simple case where an user asks to the assistant reproduce a random song. With the capability of recognizing the user's emotional state, the assistant can reproduce a funny and animated song if the user asked it with happiness, while it can reproduce a more profound and reflexive song if it has been asked with a sad voice tone.

Of course psychological and behavioural studies should be integrated in this procesess, but this is enough to give the idea that a machine can make feel an human more comfortable keeping into account the human's emotional status. This also increases the user's satisfaction with using the product and so all the involved parts in the interaction get advantage: users are happy and the company sells more products.

What can be improved in this kind of studies is the use of datasets that involve speakers speaking naturally, and not actors recorded with professional instruments. Even if this is very difficult to achieve for privacy reasons and also for labeling possibilities, this choice would make a giant step forward in speech emotion recognintion fields.

References

[1] Arya Aftab et al. "Light-SERNet: A lightweight fully convolutional neural network for speech emotion recognition". In: *arXiv preprint arXiv:2110.03435* (2021). URL: https://arxiv.org/abs/2110.03435.

[2] *GitHub: AryaAftab/LIGHT-SERNET*. URL: https://github.com/AryaAftab/LIGHT-SERNET.

[3] *GitHub: PanK0/LIGHT-SERNET*. URL: https://github.com/PanK0/LIGHT-SERNET.

[4] *GitHub: PanK0/LIGHT-SERNET/train.py*. URL: https://github.com/PanK0/LIGHT-SERNET/blob/master/train.py.

[5] D. Yu Han and I. Tashev. "Speech emotion recognition using deep neural network and extreme learning machine". In: *Proc. Annu. Conf. Int. Speech Commun. Assoc. INTERSPEECH* ().

[6] *GitHub: PanK0/LIGHT-SERNET/models.py*. URL: https://github.com/PanK0/LIGHT-SERNET/blob/master/models.py.

[7] *GitHub: PanK0/LIGHT-SERNET/utils/segment/segment_dataset.py*. URL: https://github.com/PanK0/LIGHT-SERNET/blob/master/utils/segment/segment_dataset.py.

[8] *GitHub: PanK0/LIGHT-SERNET/dataio.py*. URL: https://github.com/PanK0/LIGHT-SERNET/blob/master/dataio.py.

[9] W. Norris A. Araujo and J. Sim. "Computing receptive fields of convolutional neural networks". In: *distill, vol. 4, no. 11* (). URL: https://distill.pub/2019/computing-receptive-fields/.

[10] "IEMOCAP: Interactive emotional dyadic motion capture database". In: *Lang. Resour. Eval., vol. 42, no. 4, pp. 335–359* ().

[11] "A database of german emotional speech". In: *European Conf. Speech Commun. Technol., vol. 5, pp. 1517–1520* (). URL: http://emodb.bilderbar.info/docu/.

[12] *GitHub: PanK0/LIGHT-SERNET/hyperparameters.py*. URL: https://github.com/PanK0/LIGHT-SERNET/blob/master/hyperparameters.py.

[13] *GitHub: PanK0/LIGHT-SERNET/tflite_evaluate.py*. URL: https://github.com/PanK0/LIGHT-SERNET/blob/master/tflite_evaluate.py.

[14] *GitHub: PanK0/LIGHT-SERNET/inference_tests*. URL: https://github.com/PanK0/LIGHT-SERNET/tree/master/inference_tests.

[15] *GitHub: PanK0/LIGHT-SERNET/inference_single_file.py*. URL: https://github.com/PanK0/LIGHT-SERNET/blob/master/inference_single_file.py.

[16] *GitHub: PanK0/LIGHT-SERNET/inference_data_processing.py*. URL: https://github.com/PanK0/LIGHT-SERNET/blob/master/inference_data_processing.py.