instRecCnn Documentation

Release 1.0

Juan Sebastián Gómez Canón

May 05, 2018

CONTENTS

Juan Sebastián Gómez Cañón (Ilmenau, Germany - 2018)

CHAPTER

ONE

INTRODUCTION

In the context of growing digital media and new classification/indexing demands, the task of Automatic Instrument Recognition in the field of Music Information Retrieval (MIR) has increasing importance. Through the use of deep learning techniques, namely convolutional neural networks, and different automatic source separation algorithms, developed at the Fraunhofer Institut für Digitale Medientechnologie (IDMT)[1][2], this Master thesis investigates this recognition task and how different pre-processing stages can improve its classification performance. Several experiments have been conducted in order to reproduce and improve upon the results of the reference system reported by Han et al.[3] . Two systems are proposed in this research: an improved system using harmonic/percussive separation and post-processing using class-wise thresholding, and a combined system using solo/accompaniment separation and transfer learning methods for the special use case of jazz solo recognition. To validate the obtained results, diverse tests have been performed with multiple music data sets, with different complexities and instrument selections.

[1] Estefanía Cano, Mark D. Plumbley, and Christian Dittmar, *"Phase-based harmonic/percussive separation,"* in Proceedings of the Annual Conference of the International Speech Communication Association (INTERSPEECH), Singapore, 2014, pp. 1628-1632.

[2] Estefanía Cano, Gerald Schuller, and Christian Dittmar, *"Pitch-informed solo and accompaniment separation towards its use in music education applications,"* EURASIP Journal on Advances in Signal Processing, vol. 2014, pp. 23, 2014.

[3] Yoonchang Han, Jaehun Kim, and Kyogu Lee, *"Deep convolutional neural networks for predominant instrument recognition in polyphonic music,"* IEEE/ACM Transactions on Audio, Speech, and Language Processing, vol. 25, no. 1, pp. 208-221, Jan 2017.

CHAPTER

TWO

GETTING STARTED

2.1 Installation

This implementation uses the *librosa* python package in order to extract the melspectrograms that are used to train the neural network. *Pandas* is used to organize the results in dataframes and evaluate them. *Keras* is the neural networks API used for the implementation of all the models. Finally, *matplotlib* is used to plot the final results of the performance metrics obtained. Additionally, the implementation was created in Python 3.

2.1.1 pypi

To install the necessary packages, you can run the following command:

```
sudo pip install librosa, numpy, pandas, keras, matplotlib
```

2.1.2 TensorFlow

Please refer to this link to install TensorFlow depending on your operating system.

2.2 Settings

The file *settings.py* contains all the parameters that have been used based on the paper by Han et al. You can edit this file to the desired parameters, data sets, and paths.

2.2.1 Parameters

- **time_duration:** duration of spectrogram patches in seconds (default: 1).
- **test_overlap:** overlap used when extracting the melspectrograms of the testing data sets (default: 0.5).
- **num_mel_bands:** number of mel bands to use in the linear to mel frequency transformation (default: 128).
- **nb_epochs:** maximum number of epochs to perform during training (default: 400).
- **batch_size:** mini-batch size (default: 128).
- **learning_rate:** learning rate used for optimization during backpropagation (default: 0.001).
- **alpha:** alpha parameter used for leaky rectified linear units (LReLU) (default: 0.33).
- **iter_num:** number of training experiments to perform and evaluate (default: 3).

- **nb_classes:** number of classes for each data set.
- **val_split:** train-validation split (default: 0.85).

2.2.2 Platform

By using the *platform* python package, this script automatically selects the corresponding path for the data sets and the extracted features.

2.2.3 Create Metadata

To extract the metadata of the data set and create CSV files, run the command:

```
python3 create_encoding.py
```

At this moment, it is necessary to select the data set to be used during training and testing of the convolutional neural network. Since several experiments have been conducted, please refer to the thesis in sections **5.4 - 5.7** (Proposed Experiments). The currently available data sets are the following:

- [-2] **Jazz-Solo** - Jazz data set pre-processed with the solo-accompaniment source separation algorithm *(solo component)*.
- [-1] **JazzDB** - Jazz data set with no pre-processing.
- [0] **IRMAS Multires** - IRMAS data set with three dimensional melspectrograms as input (note: this data set must be used with the *model_multi_res* network).
- [1] **IRMAS** - IRMAS data set with no pre-processing.
- [2] **Monotimbral** - Monotimbral data set with no pre-processing.
- [3] **IRMAS Harm** - IRMAS data set pre-processed with the harmonic-percussive source separation algorithm *(harmonic component)*.
- [4] **Monotimbral Harm** - Monotimbral data set pre-processed with the harmonic-percussive source separation algorithm *(harmonic component)*.
- [5] **Monotimbral Perc** - Monotimbral data set pre-processed with the harmonic-percussive source separation algorithm *(percussive component)*.
- [6] **IRMAS Wind Solo** - IRMAS Wind data set pre-processed with the solo-accompaniment source separation algorithm *(solo component)*.
- [7] **IRMAS Wind** - IRMAS Wind data set with no pre-processing.
- [8] **IRMAS Harm-Perc** - IRMAS data set pre-processed with the harmonic-percussive source separation algorithm *(harmonic and percussive components)* (note: this data set must be used with the *model_two_branch* network).
- [9] **Monotimbral Harm-Perc** - Monotimbral data set pre-processed with the harmonic-percussive source separation algorithm *(harmonic and percussive components)* (note: this data set must be used with the *model_two_branch* network).

Please ignore the second selector at this stage.

CHAPTER

THREE

DATA SETS AND TRAINING

3.1 Generate data set

3.1.1 Usage

To extract the features for a data set and save the resulting tensors, make sure that the ground truth metadata has been previously created by using the *create_encoding.py* script.

Given the amount of processing, this class uses the *multiprocessing* package and it is recommended to run on a GPU, using the following command:

```
python3 generate_dataset.py
```

3.1.2 Documentation

class generate_dataset.**GenerateDataset**(*duration*, *num_classes*, *val_split*)

GenerateDataset generates training and testing tensors for a given dataset. The training inputs to the neural network are melspectrograms with 128 mel bands and variable segment duration length. This class also performs a balanced train-validation split depending on the amount of samples in each class. All of the parameters are inherited from *settings.py*.

Parameters

- **duration** (*int*) – number of time bins to create melspectrogram.
- **num_classes** (*int*) – number of classes to be evaluated.
- **val_split** (*float*) – training-validation split from 0 to 1.

create_multi_spectrogram(*filename*, *sr=22050*, *win_length=1024*, *hop_length=512*, *num_mel=128*)

This method creates a melspectrogram from an audio file using librosa audio processing library. Parameters are default from Han et al. It also extracts three spectrograms with different window sizes (multiples of the original window size) and stacks them into a three-dimensional representation of the audio.

Parameters

- **filename** (*str*) – wav filename to process.
- **sr** (*int*) – sampling rate in Hz (default: 22050).
- **win_length** (*int*) – window length for STFT (default: 1024).
- **hop_length** (*int*) – hop length for STFT (default: 512).
- **num_mel** (*int*) – number of mel bands (default:128).

Returns **ln_S** *(np.array)* - melspectrogram of the complete audio file with logarithmic compression with dimensionality [mel bands x time frames x 3].

`create_spectrogram` (*filename*, *sr=22050*, *win_length=1024*, *hop_length=512*, *num_mel=128*)

This method creates a melspectrogram from an audio file using librosa audio processing library. Parameters are default from Han et al.

Parameters

- **`filename`** (`str`) – wav filename to process.
- **`sr`** (`int`) – sampling rate in Hz (default: 22050).
- **`win_length`** (`int`) – window length for STFT (default: 1024).
- **`hop_length`** (`int`) – hop length for STFT (default: 512).
- **`num_mel`** (`int`) – number of mel bands (default:128).

Returns **ln_S** *(np.array)* - melspectrogram of the complete audio file with logarithmic compression with dimensionality [mel bands x time frames].

`load_metadata` (*path_metadata*)

This method loads the metadata from the dataset previously generated by *create_encoding.py*.

Parameters **`path_metadata`** (`str`) – path to csv with filenames and labels.

Returns

- **filenames** *(list)* - list of filenames that exist in the metadata.
- **labels** *(list)* - list of labels per filename that exist in the metadata.

`pure_development_split` (*dataset*, *file_id*)

This method creates balanced 50-50 split between pure testing data and development testing data from the test data set, depending on the total number of examples from each class. This method is only available for testing data sets and can manage to output a single tensor (for the normal model - multi_input = **False**) and double tensors (for the model with two branches - multi_input = **True**).

Parameters

- **`dataset`** (`list`) – list of extracted spectrograms for test data set.
- **`file_id`** (`list`) – file identifier from which the spectrogram was extracted.

Returns

- **pure_test_set** *(np.array)* - array(s) with spectrograms of pure test data set.
- **fid_pure_set** *(list)* - list(s) of file ids for the pure test data set.
- **dev_test_set** *(np.array)* - array(s) with spectrograms of development test data set.
- **fid_dev_set** *(list)* - list(s) of file ids for the development test data set.

`run` (*type_run*)

This method extracts the features for the training, validation, development test, and pure test data sets and saves the resulting tensors to train and evaluate the convolutional neural network. It uses the *multiprocessing* package and the amount of processes created depends on the quantity of multiprocessing.cpu_count() function. It has been created for the normal model (multi_input = **False**).

Parameters **`type_run`** (`str`) – generates 'train' or 'test' data sets.

`run_multi` (*type_run*)

This method extracts the features for the training, validation, development test, and pure test data sets and saves the resulting tensors to train and evaluate the convolutional neural network. It uses the *multiprocessing* package and the amount of processes created depends on the quantity of multiprocessing.cpu_count() function. It has been created for the two branch model (multi_input = **True**).

Parameters **`type_run`** (*`str`*) – generates 'train' or 'test' data sets.

`validation_split` (*dataset*, *class_id*, *file_id*, *num_classes*, *validation*)

This method shuffles the samples randomly and creates a balanced train-validation split depending on the total number of examples from each class. This method is only available for training data sets and can manage to output a single tensor (for the normal model - multi_input = **False**) and double tensors (for the model with two branches - multi_input = **True**).

Parameters

- **`dataset`** (*`list`*) – list of extracted melspectrograms for the dataset.
- **`class_id`** (*`list`*) – list of labels from the extracted spectrograms.
- **`file_id`** (*`list`*) – file identifier from which the spectrogram was extracted.
- **`num_classes`** (*`int`*) – number of classes in the dataset.
- **`validation`** (*`float`*) – train-validation split from 0 to 1.

Returns

- **X_train** *(np.array)* - array(s) with the training data set of spectrograms.
- **y_train** *(np.array)* - array(s) with training labels of training data set.
- **fid_train** *(list)* - list(s) of file ids for the training data set.
- **X_val** *(np.array)* - array(s) with validation data set of spectrograms.
- **Y_val** *(np.array)* - array(s) with validation labels of validation dataset
- **fid_val** *(list)* - list(s) of file ids for the validation data set.

3.2 Train the Neural Network

3.2.1 Usage

After creating the tensors that will be used to train and test the neural network, it is possible to start training the neural network. Different models and optimizers can be selected by command-line argument parsing. Additionally, the *class_weight* implementation from *Keras* can be activated. This was implemented to take into account the uneven distribution of samples across instrument labels. Given that this did not result in improvements with the IRMAS data set, the use of this feature is optional in argument parsing. *Since the size of the neural network and the amount of data, it is strongly recommended to use a GPU to reduce training time.*

Available model names are:

- **model_baseline** - Baseline model by Han et al. using ReLU activation functions (refer to **section 4.1**).
- **model_leaky** - Baseline model by Han et al. using Leaky ReLU activation functions (refer to **section 4.1**).
- **model_two_branch** - Model that allows for two simultaneous inputs and late fusion (refer to **section 5.6.2**).
- **model_multi_res** - Model that allows for three dimensional input (refer to **section 5.6.3**).

Available optimizer names are:

- **adam** - Adam optimizer with parameters beta_1=0.9, beta_2=0.999, epsilon=1e-08.

- **sgd** - Stochastic Gradient Descent optimizer with parameters momentum=0.9, decay=learning_rate/nb_epochs, Nesterov=True.

To train the network, use the following command:

```
python3 train.py -m <model_name> -o <optimizer_name> -c <y/n>
```

Finally, the resulting files are stored in the MODEL_PATH defined in *settings.py*:

- **<model_name>.<optimizer_name><iteration>.history.npy** - Training history to evaluate training and validation loss and accuracy.
- **<model_name>.<optimizer_name><iteration>.best.hdf5** - Stored weights by using ModelCheckpoint saving the model with best validation accuracy.
- **<model_name>.<optimizer_name><iteration>.json** - Structure of the neural network.

3.2.2 Documentation

class train.**Trainer**(*model*, *optimizer*, *data_dist*)

Trainer trains a convolutional neural network with a given model. The training data set must have already been extracted with *generate_dataset.py*. The balanced train-validation split is already performed during the data set generation, so the trainer only performs a randoms shuffling on every epoch. All of the parameters are inherited from *settings.py*.

Parameters

- **model** (*object*) – model to be trained with the selected data set.
- **optimizer** (*str*) – optimizer to be used in backpropagation.
- **data_dist** (*pandas dataframe*) – dataframe with samples distribution in dataset (if class_weight is implemented).

load_data()

This method loads the training and validation tensors (melspectrogram arrays) and corresponding labels to memory (for the case of single input). It also implements one hot encoding for the labels (both training and validation data sets).

load_multi_data()

This method loads the training and validation tensors (melspectrogram arrays) and corresponding labels to memory (for the case of multiple input). It also implements one hot encoding for the labels (both training and validation data sets).

train(*epochs*, *batch*, *iter_num*)

This method trains the corresponding model and implements two callback functions: Early Stopping (patience = 5 epochs) and Model Checkpointing. The number of epochs and mini-batch size are inherited from *settings.py*. Additionally, the training history, weights, and structure are stored in the MODEL_PATH.

Parameters

- **epoch** (*int*) – maximum number of epochs to train model.
- **batch** (*int*) – mini-batch size.
- **iter_num** (*int*) – number of training iterations to compensate for random initialization.

CHAPTER

FOUR

EVALUATION AND RESULTS

4.1 Evaluate the Performance

4.1.1 Usage

After creating the tensors that will be used to train and test the neural network, it is possible to start training the neural network. Different models and optimizers can be selected by command-line argument parsing. Additionally, the **class_weight** implementation from *Keras* can be activated. This was implemented to take into account the uneven distribution of samples across instrument labels. Given that this did not result in improvements with the IRMAS data set, the use of this feature is optional in argument parsing.

Available models are: **model_baseline**, **model_leaky**, **model_two_branch**, and **model_multi_res**. Available optimizers are: **adam** and **sgd**

To train the network, use the following command:

```
python3 evaluate.py -m <model_name> -o <optimizer_name> -O <1/2/3>
```

Depending on the operation mode, follow the instructions to select the split.

4.1.2 Operation Mode 1

This operation mode is used to extract confusion matrices. Since confusion matrices are only valid for single labeled data, there are two possibilities depending on the used data sets:

- **IRMAS and IRMAS Wind data sets:** since these data sets are multi-labeled on the testing data, a new model is trained by using 50% of the training data set and the remain for confusion matrix calculation.
- **Monotimbral and Jazz data sets:** since these data sets are single-labeled in both training and testing data sets, confusion matrices can be extracted by using the pure test data set.

To plot the confusion matrices:

```
cd <MODEL_PATH>/<model_name>
python3 plot_conf_mat.py
```

4.1.3 Operation Mode 2

This operation mode evaluates global and class-wise performance metrics while varying the identification threshold from 0 to 1. Following the authors, the development test data set was used to calculate the performance metrics.

To be able to extract the final performance metrics, it is necessary to plot their behaviour with respect to the identification threshold:

```
cd <MODEL_PATH>/<model_name>
python3 plot_perf_metrics.py
```

Do not skip this step, before extracting final performance metrics.

4.1.4 Operation Mode 3

Finally, the final performance metrics are extracted by using the optima identification thresholds (for both global and class-wise cases) on the pure test data set.

4.1.5 Documentation

class `evaluate.`**`Evaluator`**(*model_str*, *optimizer_str*, *num_classes*, *iter_num*, *op_mode*)

Evaluator uses the testing data set to reproduce results from the original neural network design and evaluate the improvements made by audio source separation and different proposed experiments. This testing algorithm allows variable length of audio excerpts.

Parameters

- **`model`** (*`str`*) – model to be evaluated.
- **`optimizer`** (*`str`*) – optimizer that was used in training.
- **`num_classes`** (*`int`*) – number of classes that were trained in the model.
- **`iter_num`** (*`int`*) – number of training iterations to evaluate.
- **`op_mode`** (*`int`*) – operation mode [1] evaluate confusion matrix, [2] evaluate classwise and global performance metrics and [3] final performance metrics using the pure test data set.

`aggregate_predictions`(*strategy*, *full_predictions*)

This method performs different aggregation strategies to obtain a final prediction for each sample (complete audio excerpt) in the test data set.

Parameters

- **`strategy`** (*`str`*) – name of the aggregation strategy to evaluate.
- **`full_predictions`** (*`list`*) – complete predictions over all excerpts in test data set.

`best_classwise_global_performance_metrics`(*strategy*, *iteration*)

This method calculates the global performance metrics of the system, using a novel approach: using the optima class-wise thresholds in in the pure test data set. It calculates precision, recall and f-score for micro and macro averaging. It also evaluates different performance metrics using a variable threshold for each instrument separately.

Parameters

- **`strategy`** (*`str`*) – name of the aggregation strategy to store.
- **`iteration`** (*`int`*) – id of the iteration being evaluated.

`classwise_performance_metrics`(*strategy*, *iteration*)

This method calculates the classwise performance metrics of the system. It calculates precision, recall and f-score for micro and macro averaging. It also evaluates different performance metrics using a variable threshold for each instrument separately.

Parameters

- **`strategy`** (*`str`*) – name of the aggregation strategy to store.
- **`iteration`** (*`int`*) – id of the iteration being evaluated.

`evaluate`(*predictions*, *model_str*, *optimizer_str*, *iteration*)

This method evaluates the corresponding trained model i with different aggregation strategies for each sample in the dataset. It saves a dictionary with the corresponding performance metrics depending on the operation mode of the evaluator and the incoming predicted values.

Parameters

- **`predictions`** (*`list`*) – calculated predictions for model i and selected operation mode.
- **`model_str`** (*`str`*) – name of the model to load.
- **`optimizer_str`** (*`str`*) – name of the optimizer used in training.
- **`iteration`** (*`int`*) – id of the iteration being evaluated.

`global_performance_metrics`(*strategy*, *iteration*)

This method calculates the global performance metrics of the system. It calculates precision, recall and f-score for micro and macro averaging. It also evaluates different performance metrics using a variable global threshold for all instruments.

Parameters

- **`strategy`** (*`str`*) – name of the aggregation strategy to store.
- **`iteration`** (*`int`*) – id of the iteration being evaluated.

`load_model`(*model_str*, *optimizer_str*)

This method loads the model and the weights obtained in training. Each model is represented by the activation function used and the optimizer used in backpropagation. It also calculates the predictions for every melspectrogram in the testing data sets (validation, development, and pure test data sets).

Parameters

- **`model_str`** (*`str`*) – name of the model to load.
- **`optimizer_str`** (*`str`*) – name of the optimizer used in training.

`load_multi_test_data`()

This method loads the multiple-input development test data (melspectrogram arrays) and corresponding file ids to memory. It also implements one hot encoding and multilabel binarizer for the labels in the test data set. The loaded data set depends on the operation mode selected to perform the evaluation: [1] loads 50-50 train-validation split, [2] loads variable train-validation split, [3] loads pure testing data set.

`load_test_data`()

This method loads the single-input development test data (melspectrogram arrays) and corresponding file ids to memory. It also implements one hot encoding and multilabel binarizer for the labels in the test data set. The loaded data set depends on the operation mode selected to perform the evaluation: [1] loads 50-50 train-validation split, [2] loads variable train-validation split, [3] loads pure testing data set.

`save_results`(*model_str*, *optimizer_str*)

This method saves the results of the evaluation depending on the operation mode of the evaluator.

Parameters

- **`model_str`** (*`str`*) – name of the model
- **`optimizer_str`** (*`str`*) – name of the optimizer

4.2 Comparing Results

4.2.1 Usage

The final results obtained by the evaluation script can now be compared among all trained models. To do this interactively, use the following commands:

```
cd models
ipython3 compare_models.py
```

4.2.2 Review Results

The results of performance metrics are stored to Pandas data frames. Results are stored in the following *Pandas* dataframes:

- **global_res:** Global performance metrics.
- **class_res:** Class-wise performance metrics.
- **global_class_res:** Global performance metrics by using optima class-wise thresholds.

By accessing it interactively, it is possible to find the global performance metrics with respect to the data set, model, aggregation strategy and averaging method used.

For example:

```
global_res.loc['irmas_1024'].loc['model_baseline'].loc['s2'].loc['micro']
class_res.loc['jazz_db_1024'].loc['model_baseline'].loc['s1'].loc['voi']
global_class_res.loc['youtube_1024'].loc['model_leaky'].loc['s1'].loc['macro']
```

will print the precision, recall, and f-scores of these cases.

Finally, you can uncomment sections of the script to plot the improvements by comparing the models in between them.

CHAPTER

FIVE

REFERENCES

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX