Experiments Document

SOURCE CODE STRUCTURE

As shown in Fig.1, the root path `SED_MM/` of source code is made up of four folders: `aed_data/`, `MainClasse/`, `model_figures/`, and `task_scripts/`. The detailed introduction about them is given in the following.

- `SED_MM/aed_data/`

This folder contains the datasets (i.e., Freesound and TUT), together with the experiment results on these two datasets. Both under the database directories of `freesound/` and `tut_data/`, there are two folders `mfccs/` and `result/` respectively, which are shown in Fig.1. Specifically, `mfccs/` provides the input raw data, and `result/` saves the results of each experiment on the corresponding database. The detailed folder structure of `aed_data/` is described in the following.

- `freesound/`
  - `ae_dataset/`: saves the original Freesound dataset.
  - `audio_target/`: saves the discriminative audio signal which are selected from the original Freesound dataset.
  - `audio/`: saves the short sound segmants cut from `audio_target/` via Audacity (a free software). Here, the labels of short sound segments are the same as the label of the long audio signal that they are cut from.
  - `mfccs/`:
  - `datas/`: contains MFCC features (`*.pkl` file), named as `mfcc_[events number]_.pkl`).
  - `labels/`: contains pickle files which are corresponding to feature files.
  - `noise/`: contains three types of environment noise, which will be used when generating sub-set with different polyphonic level.
  - `result/`: saves the model weights in the training stage and the results of cross evaluation as table files (`*.csv`).
- `tut_data/`
  - `train_val_split/`: There are four text files which are used to generate the training set and validation set with the 4-folds cross evaluation.
  - `train/`: saves the data related to the training set, which includes annotation files, original signal files and MFCC features.
  - `test/`: saves the testing set.
  - `result/`: saves the model weights and experiment results.
- `SED_MM/MainClasses/`

This folder contains several **core files** (Python classe files), which define the data processing, models initialization, the training and evaluation process. The relationship between these classes and task scripts are shown in Fig.2. The details about each class are given below.

- `Dataset.py`: This Python script defines the class of parameter initialization and data processing, which contain a model initialization function and several methods for data preprocessing, such as `enframe()`, `windowing()`, etc. Besides, it provides all the raw input data for each task script by `load_data()` method. Note that, the most important parameter of `load_data()` is `fold`, which is used to generate the training and validation set. It is worth mentioning that the method `mix_data()` is essential for the Freesound dataset, because it is used to generate the sub-dataset with various polyphonic levels. Here is the usage of mixing the polyphonic sound:
  1.Load `Dataset` class and create an object:

```
from Dataset import *
dt = Dataset('path/to/dataset')
```

2.Generate `n` mixture sound with `k` categories, and here `k` denotes polyphonic level:

```
dt.mix_data(nevents=k, nsamples=n)
```

By using `mix_data()`, we can get a `.pkl` file which contains the MFCCs features extracted from mixed sound.

- `Models.py`: This Python script defines the super class `Model`, which includes `train_model()` and `metric_model()` functions. Specifically, the `train_model()` function serves the training stage and saves the weights of model, while the `metric_model()` defines two metrics (segment based F1 score and Error Rate), which are totally the same as metrics used in the DCASE2017 AED task. By extending the class `Model`, we can easily build different networks.
- `AttSBetaVAE.py`: This Python script is the **core file of the archives** which extends the super class `Model` with the `build_model()` method. Specifically, in `AttSBetaVAE.py`, the first nested function `sampling()` calculates the latent factors `z` by Gaussian Sampling. The `loss_disent()` defines the proposed novel disentangling loss as Eq.6 in the accepted paper. In `loss_disent()`, `beta` is normalized by using the same method as the original beta-VAE. `R_square()` function is given to evaluate the reconstruction performance of our proposed method. The major layers' definition of our framework proposed in paper are given following the `R_square()`, which builds encoder, latent attention, de-coder and event detector of our network respectively. The last nested function `generate_data()` helps to generate new data for specific sound event. To build and compile the proposed model, the Python codes below are needed:

  ```
  model_generator = AttSBetaVAE()
  model = model_generator.build_model()
  ```

- `SED_MM/model_figures`

This folder saves the figures of model summary, which clearly show the structure of models. To plot the model figure automatically, the `plot_figure` module must be first imported from `keras.utils`. Then, after executing the Python code below, we can get the `fig.png`.

```
plot_model(model, to_file='model_figures/fig.png', show_shapes=True)
```

- `SED_MM/task_scripts/`

  This folder contains Python scripts provided for the major experiments implemented in the accepted paper. **It is important to review the codes carefully for the replicating work**. We provide five scripts for various tasks mentioned in the accepted paper. To make these scripts easier to read, we reseal each script with three functions: `setup_args()` for arguments setup, `running()` for preparing input data, building model and executing training/evaluation process, and `clear_up()` given for memory and session clearance to avoid interfering next experiment. All the task scripts can be implemented by the following shell command, where the optional arguments can be found in `setup_args()` or `-h` in terminal:

  ```
  cd AED_MM
  python3 tast_scripts/any_script.py [args] # -h for help
  ```

  Although such command can be used to repeat the experiments, it is necessary to give a detailed introduction for each task script. **It is important to emphasize that we need execute these task scripts in the order listed below since their results are dependent with each other.**

  - `tb4_various_events_ours.py`: This script evaluates the performance of our methods on data of various polyphonic levels made from Freesound as shown in Table 4 in the accepted paper. In this script, there are many optional arguments which will determine the structure of the model. We can conduct this experiment after we choose suitable value for each optional argument. For example, if we want to train our model with 3000 samples which contain 10 event categories, and in this case, the latent factors, the hyper-parameter `beta` and `lambda` are set as 30, 4 and 2 respectively. We just need call the shell command below to conduct this evaluation experiment:

    ```
    python3 tb4_various_events_ours.py --num_samples 3000 --num_events 10
    --beta 4 --lambda 2 -z 30
    # for larger num_events, we need to generate more training samples
    with larger num_samples.
    ```

    Here, It is important to note that the argument `--mix_data` denotes whether we need generate new data, and it is set as 1 as default if you want generate new data, and when you do not need to generate new data, set as 0. The details of optional arguments can be found in `setup_args()`.

  - `fig4_disentanglement_visualization.py`: This script is used to evaluate the disentanglement performance, mentioned in Fig.4 in the accepted paper. To qualitatively show event-specific disentangled factors learned by supervised beta-VAE, we need to execute the operations below.

1. Create model and load weights;
2. Give `n` samples of input data `x` , and extract `z*` for some specific sound event categories:

```
# `i` denotes the layer index of a certain latent factor z* in the models, which can be found in the end of model.summary()
z_star_fnc = K.Function([model.input], [model.layers[i].output])
# here `x` denotes the input raw features
z_star = e_fnc([x])[0]
```

3. Adjust one latent variable while fixing others in `z_star` and visualize the corresponding changes in the generated data. Take `Children` (shown in Fig.4. in the accepted paper) as an instance, we adjust the value of the 14-th dimension of `z_star` and fixing others:

```
# caculate the min, max and middle value of the 14-th factor of n samples
min_z = z_star[:, 14].min()
max_z = z_star[:, 14].max()
mid_z = (min_z + max_z) /2.0
# define the decoder
decoder_fnc = K.Function([model.layers[6].output], [model.output[0]])
# generate new data using the changed z_star
# (1) set the value of the 14-th factor as the middle value
z_star[:, 14] = mid_z
generated_data_mid = decoder_fnc([z_star])[0]
# (2) set the value of the 14-th factor as the min value
z_star[:, 14] = min_z
generated_data_min = decoder_fnc([z_star])[0]
# (3) set the value of the 14-th factor as max value
z_star[:, 14] = max_z
generated_data_max = decoder_fnc([z_star])[0]
```

4. Define `delta()` function, mentioned in Section 4.5 of our paper, and calculate the difference among generated data:

```
def delta(z_star, factor, initial, altered):
 decoder_fnc = K.Function([model.layers[6].output], [model.output[0]])
 z_star[:, factor] = initial
 generated_datas_initial = decoder_fnc([z_star])
 z_star[:, factor] = altered
 generated_datas_altered = decoder_fnc([z_star])
 return generated_datas_altered - generated_datas_initial
# calculate the difference between the max and the middle value of
z_star[:, 14]
delta_max2mid = delta(z_star, 14, max_z, mid_z)
# calculate the difference between the middle and the min value of
z_star[:, 14]
delta_mid2min = delta(z_star, 14, mid_z, min_z)
```

5. Visualize the differences calculated above using hot figure:

```
import matplotlib.pyplot as plt
plt.imshow(delta_max2mid)
plt.imshow(delta_mid2min)
plt.colorbar()
plt.show()
```

- `tb5_data_augmentation.py`: This script gives the method to evaluate the data generation ability of the proposed method.

1. We first make up the unbalanced dataset from Freesound using the class `Dataset` mentioned before:

```
from Dataset import *
dt = Dataset('aed_data/freesound/')
```

2. Then call `mix_data()` method with the argument `isUnbalanced=True`, by which we can limit the number of the samples in the unbalanced dataset for a specific event category.

```
dt.mix_data(nevents=5, nsamples=2000, isUnbalanced=True)
```

3. After getting the unbalanced dataset, we train and evaluate the model:

```
model = att_s_beta_vae.build_model(options)
att_s_beta_vae.train_model(model, x_train=sequential_train_datas,
y_train=train_labels)
# here we get the original results
f1_score, error_rate = att_s_beta_vae.metric_model(model,
sequential_test_datas, test_labels, supervised=True,
new_weight_path='last_weight.h5')
```

4. Next, we generate the samples for the event category with insufficient samples by decoding the specific latent factors `z*`:

```
# i denotes the layer index of a certain latent factor z* in the models, which can be found after model.summary() is executed.
z_star_fnc = K.Function([model.input], [model.layers[i].output])
# here x denotes the input raw features
z_star = z_star_fnc([x])[0]
# define the decoder
decoder_fnc = K.Function([model.layers[6].output], [model.output[0]])
generated_data = decoder_fnc([z_star])[0]
```

5. At last, we extend the unbalanced dataset with the generated data to retrain the model, and evaluate it again, which will improve the performance of the augmented event category.

- `tb3_dcase17_ours.py`: This script evaluation the performance of the proposed method on DCASE 2017 SED challenge dataset `tut_data` as mentioned in Table 3 in the accepted paper. In `tb3_dcase17_ours.py`, `running()` function defines the process of building model, loading weights and training models at the 4-folds cross-validation strategy, and then saving the evaluation results into the table file (`.csv`).
- `fig3_feature_distribution.py`: This script is used to visualize and compare the distributions of the features, mentioned in Fig.3 in the accepted paper, learned by our model, respectively. It is dependent on t-SNE which should be imported from `sklearn.manifold`. In general, the distribution visualization can be divided into 5 steps:

1. Create model and load weights:

```
model_generator = AttSBetaVAE(args)
model = model_generator.build_model()
model.load_weights('path/of/weights')
```

2. Give `n` samples as input data `x`, and extract the output of hidden layers(`m_th`):

```
import keras.backend as K
# it returns a list
h_fnc = K.Function([model.input], [model.layers[m].output])
m_hidden_output = h_fnc([x])[0]
```

3. Create t-SNE object and initialize with `PCA`:

```
from sklearn.manifold import TSNE
tsne = TSNE(n_components=target_dim, init='pca')
```

4. Train and transform the high-level features into 2 dimensions:

```
tsne_datas = tsne.fit_transform(datas)
```

5. Using `matplotlib` to plot the transformed data with colorful legend labels:

```
nums_each_ev = int(len(datas) / n_events)
for i in range(n_events):
  plt.scatter(tsne_datas[nums_each_ev * i:nums_each_ev * (i + 1), 0],
              tsne_datas[nums_each_ev * i:nums_each_ev * (i + 1), 1],
              c=plt.cm.Set1((i + 1) / 10.),
              alpha=0.6, marker='o')
  plt.legend(label_list, loc='lower right', ncol=2)
  plt.title('Distribution of features learned by
{}.'.format(model.name))
  plt.show()
```

In order to simplify the procedure, in `fig3_feature_distribution.py`, we has put all the five steps into `running()` function. For more details, you need read the code comments carefully.

TIME SPENDING

At last, the runing time of the main task srcipts executed in our GPU server shown below:

`tb4_various_events_ours.py` with various events categories:

5 events with 2000 samples: 26s * 80 epochs * 4 folder;
10 events with 3000 samples: 44s * 80 epochs * 4 folder;
15 events with 4000 samples: 50s * 80 epochs * 4 folder;
20 events with 5000 samples: 73s * 80 epochs * 4 folder;

`tb3_dcase17_ours.py`:
dcase: 179s * 200 epochs * 4 folder

And the CPU of our server is Intel(R) Core(TM) i9-9820X CPU @ 3.30GHz adn the GPU is NVIDIA RTX 2080Ti.