项目 5

PROJECT 5

歌曲人声分离

本项目通过 TensorFlow 构建 Bi-LSTM，针对唱歌软件的基本伴奏资源，使用 STFT（Short Time Fourier TransForm，短时傅里叶变换）进行处理，实现原曲获得音频质量较好的伴奏和纯人声音轨。

5.1 总体设计

本部分包括系统整体结构图和系统流程图。

5.1.1 系统整体结构图

系统整体结构如图 5-1 所示。

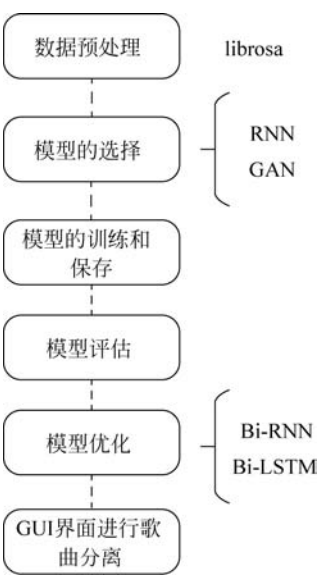

图 5-1 系统整体结构图

5.1.2 系统流程图

系统流程如图 5-2 所示。

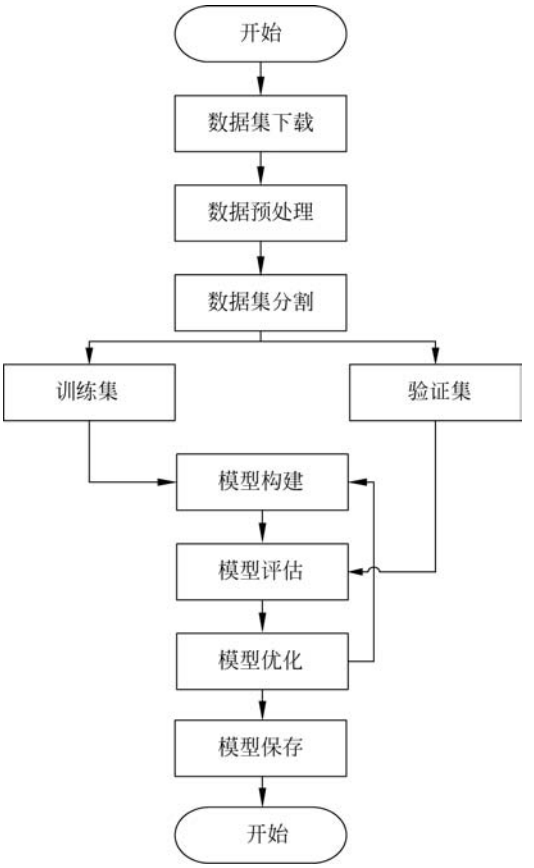

图 5-2 系统流程图

5.2 运行环境

本部分包括 Python 环境、TensorFlow 环境和 Jupyter Notebook 环境。

5.2.1 Python 环境

需要 Python 3.6 及以上配置,在 Windows 环境下推荐下载 Anaconda 完成 Python 所需的配置,下载地址为 https://www.anaconda.com/,默认下载 Python 3.7 版本。打开 Anaconda Prompt,安装开源的音频处理库 librosa,输入命令:

```
pip install librosa
```

安装完毕。

5.2.2　TensorFlow 环境

(1) 打开 Anaconda Prompt,输入清华仓库镜像,输入命令:

```
conda config -- add channels https://mirrors.tuna.tsinghua.edu.cn/anaconda/pkgs/free/
conda config - set show_channel_urls yes
```

(2) 创建 Python 3.6 环境,默认 3.7 版本和低版本 TensorFlow 存在不兼容问题,所以建立 Python 3.6 版本,输入命令:

```
conda create -n python36 python=3.6
```

依据给出的相关提示,逐步安装。

(3) 在 Anaconda Prompt 中激活创建的虚拟环境,输入命令:

```
activate python36
```

(4) 安装 CPU 版本的 TensorFlow,输入命令:

```
conda install - upgrade -- ignore-installed tensorflow
```

安装完毕。

5.2.3　Jupyter Notebook 环境

(1) 首先打开 Anaconda Prompt,激活安装 TensoFlow 的虚拟环境,输入命令:

```
activate python36
```

(2) 安装 ipykernel,输入命令:

```
conda install ipykernel
```

(3) 将此环境写入 Jupyter Notebook 的 Kernel 中,输入命令:

```
python -m ipykernel install -- name python36 -- display-name "tensorflow(python36)"
```

(4) 打开 Jupyter Notebook,进入工作目录后,输入命令:

```
jupyter notebook
```

安装完毕。

5.3 模块实现

本项目包括 5 个模块：数据准备、数据预处理、模型构建、模型训练及保存、模型测试，下面分别给出各模块的功能介绍及相关代码，目录结构如图 5-3 所示。

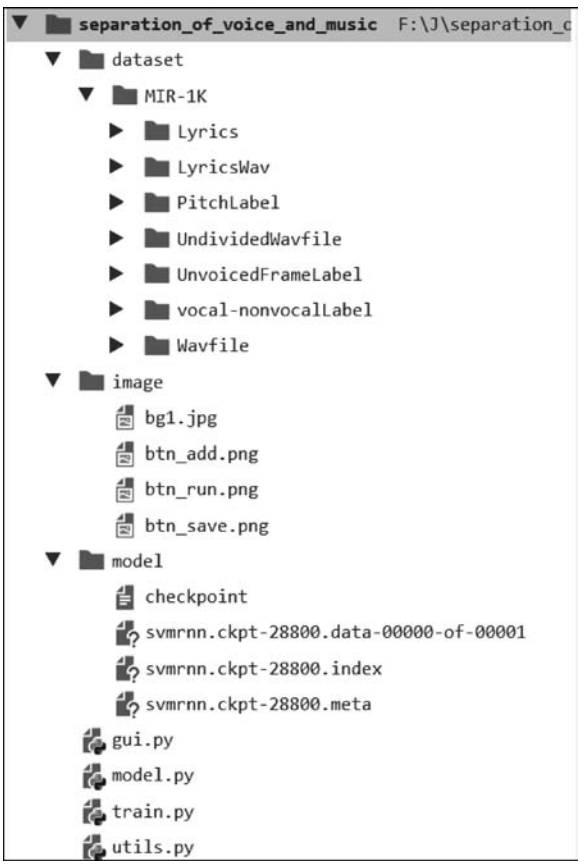

图 5-3 目录结构

(1) dataset：放置训练数据和验证数据。
(2) image：放置 GUI 制作所需要的图片素材。
(3) model：存储训练的模型。
(4) gui.py：模型的使用和 GUI 界面。
(5) model.py：构建网络模型。
(6) train.py：训练文件，包括数据的预处理、模型的训练及保存。
(7) utils.py：存放需要的工具函数。

5.3.1　数据准备

数据集适用于歌曲旋律提取、歌声分离、人声检测，如图 5-4 所示。

Wavfile 文件夹包含 1150 个歌曲文件，全部采用双声道.wav 格式。左声道为人声，右声道为伴奏。UndividedWavfile 文件夹中包含了 110 个歌曲文件，文件格式为单声道.wav。

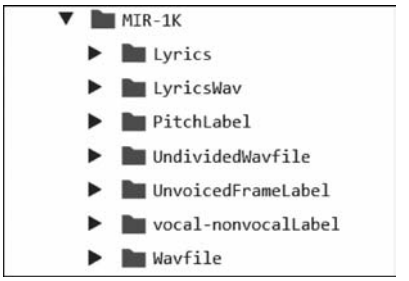

图 5-4　数据集结构

Wavfile 文件夹中的所有数据作为训练集，UndividedWavfile 文件夹中的所有数据作为验证集。将 MIR-1K 下载解压至工作目录中的 dataset 文件夹中，完成数据集的准备工作。数据集下载地址为 http://mirlab.org/dataset/public/MIR-1K_for_MIREX.rar。

5.3.2　数据预处理

数据预处理主要完成如下功能：载入文件列表；将列表中的.wav 文件读取到内存中，并分解为单声道原曲、单声道伴奏、单声道纯人声；把处理好的音频进行短时傅里叶变化。具体包括载入数据集和验证集文件列表、读入文件并进行分离、.wav 文件进行 STFT 变换、处理频域文件、相关路径与参数的设定。

1. 载入数据集和验证集文件列表

将 dataset 文件夹中的训练集和验证集文件建立索引，保存到列表中，进行数据预处理。

```
#导入操作系统模块
import os
#导入文件夹中的数据
def load_file(dir):
    file_list = list()
    for filename in os.listdir(dir):
        file_list.append(os.path.join(dir, filename))
  #将文件夹中的信息存入 file_list 列表中
    return file_list
#设置数据集路径
dataset_train_dir = './dataset/MIR-1K/Wavfile'
dataset_validate_dir = './dataset/MIR-1K/UndividedWavfile'
train_file_list = load_file(dataset_train_dir)
valid_file_list = load_file(dataset_validate_dir)
```

2. 读入文件并进行分离

将列表中的文件读取到内存中，进行音频的分离操作。原数据集中的音频文件都是双声

道文件。其中左声道为纯伴奏声轨,右声道为纯人声轨,这里将左右声道和原文件分别保存。

```
#导入音频处理库
import librosa
#数据集的采样率
mir1k_sr = 16000
#将 file_list 中的文件读入内存
def load_wavs(filenames, sr):
    wavs_mono = list()
    wavs_music = list()
    wavs_voice = list()
    #读取.wav 文件(要求源文件是双声道的音频文件,一个声道是纯伴奏,另一个声道是纯人声)
    #将音频转换成单声道,存入 wavs_mono
    #将纯伴奏存入 wavs_music,
    #将纯人声存入 wavs_voice
    for filename in filenames:
        #librosa.load 函数:根据输入的采样率将文件读入
        wav, _ = librosa.load(filename, sr=sr, mono=False)
        assert(wav.ndim == 2) and (wav.shape[0] == 2), '要求 WAV 文件有两个声道!'
        #librosa.to_mono:将双声道转为单声道
        wav_mono = librosa.to_mono(wav) * 2
        wav_music = wav[0, :]
        wav_voice = wav[1, :]
        wavs_mono.append(wav_mono)
        wavs_music.append(wav_music)
        wavs_voice.append(wav_voice)
    #返回单声道原歌曲、纯伴奏、纯人声
    return wavs_mono, wavs_music, wavs_voice
    #导入训练数据集的.wav 音频数据
#wavs_mono_train 存的是单声道音频,wavs_music_train 存的是纯伴奏,wavs_voice_train 存的是
纯人声
    wavs_mono_train, wavs_music_train, wavs_voice_train = load_wavs(filenames=train_file_
list, sr=mir1k_sr)
    #导入验证集的.wav 数据
    wavs_mono_valid, wavs_music_valid, wavs_voice_valid = load_wavs(filenames=valid_file_
list, sr=mir1k_sr)
```

3. wav 文件进行 STFT 变换

将读入的验证集和训练集音频文件从时域转化为频域。调用自定义的频域转换函数 wavs_to_specs()。但是训练集的数量较大,如果一次性全部转换,会导致内存占用过多。因此,每次随机抽取一个 batch_size 大小的文件进行频域转换。

```
#导入 numpy 数学库
import numpy
#通过短时傅里叶变换将声音转到频域
#三组数据分别进行转换
```

```
def wavs_to_specs(wavs_mono, wavs_music, wavs_voice, n_fft = 1024, hop_length = None):
    stfts_mono = list()
    stfts_music = list()
    stfts_voice = list()
    for wav_mono, wav_music, wav_voice in zip(wavs_mono, wavs_music, wavs_voice):
    #在 librosa0.7.1 及以上版本中,单声道音频文件必须为 fortran - array 格式,才能送入
librosa.stft()进行处理
    #使用 numpy.asfortranarray()函数,对单声道音频文件进行上述格式转换
        stft_mono = librosa.stft((numpy.asfortranarray(wav_mono)), n_fft = n_fft, hop_
length = hop_length)
        stft_music = librosa.stft((numpy.asfortranarray(wav_music)), n_fft = n_fft, hop_
length = hop_length)
        stft_voice = librosa.stft((numpy.asfortranarray(wav_voice)), n_fft = n_fft, hop_
length = hop_length)
        stfts_mono.append(stft_mono)
        stfts_music.append(stft_music)
        stfts_voice.append(stft_voice)
    return stfts_mono, stfts_music, stfts_voice
#调用 wavs_to_specs 函数,转化验证集的数据
stfts_mono_valid, stfts_music_valid, stfts_voice_valid = wavs_to_specs(wavs_mono = wavs_
mono_valid, wavs_music = wavs_music_valid, wavs_voice = wavs_voice_valid, n_fft = n_fft,hop_
length = hop_length)
#定义 batch_size 大小
batch_size = 64
#定义 n_fft 大小,STFT 的窗口大小
n_fft = 1024
#定义存储要转化训练集数据的数组
    wavs_mono_train_cut = list()
wavs_music_train_cut = list()
wavs_voice_train_cut = list()
#从训练集中随机选取 64 个音频数据
for seed in range(batch_size):
    index = np.random.randint(0,len(wavs_mono_train))
    wavs_mono_train_cut.append(wavs_mono_train[index])
    wavs_music_train_cut.append(wavs_music_train[index])
    wavs_voice_train_cut.append(wavs_voice_train[index])
#短时傅里叶变换,将选取的音频数据转到频域
stfts_mono_train_cut, stfts_music_train_cut, stfts_voice_train_cut = wavs_to_specs(wavs_
mono = wavs_mono_train_cut, wavs_music = wavs_music_train_cut, wavs_voice = wavs_voice_
train_cut,n_fft = n_fft, hop_length = hop_length)
```

4. 处理频域文件

频域文件中的数据是复数,包含频率信息和相位信息,但是在训练时只需要考虑频率信息,所以将频率和相位信息分开,使用 mini_batch 的方法进行训练数据的输入。

```
#获取频率
```

```
def separate_magnitude_phase(data):
    return np.abs(data), numpy.angle(data)
#mini_batch 进行数据的输入
#stfts_mono:单声道 STFT 频域数据
#stfts_music:纯伴奏 STFT 频域数据
#stfts_music:纯人声 STFT 频域数据
#batch_size:batch 的大小
#sample_frames:获取多少帧数据
def get_next_batch(stfts_mono, stfts_music, stfts_voice, batch_size = 64, sample_frames = 8):
    stft_mono_batch = list()
    stft_music_batch = list()
    stft_voice_batch = list()
    #随机选择 batch_size 个数据
    collection_size = len(stfts_mono)
    collection_idx = numpy.random.choice(collection_size, batch_size, replace = True)
    for idx in collection_idx:
        stft_mono = stfts_mono[idx]
        stft_music = stfts_music[idx]
        stft_voice = stfts_voice[idx]
        #统计有多少帧
        num_frames = stft_mono.shape[1]
        assert num_frames >= sample_frames
        #随机获取 sample_frames 帧数据
        start = numpy.random.randint(num_frames - sample_frames + 1)
        end = start + sample_frames
        stft_mono_batch.append(stft_mono[:,start:end])
        stft_music_batch.append(stft_music[:,start:end])
        stft_voice_batch.append(stft_voice[:,start:end])
        #将数据转成 numpy.array,再对形状做一些变换
    #Shape: [batch_size, n_frequencies, n_frames]
    stft_mono_batch = numpy.array(stft_mono_batch)
    stft_music_batch = numpy.rray(stft_music_batch)
    stft_voice_batch = numpy.array(stft_voice_batch)
    #送入 RNN 的形状要求: [batch_size, n_frames, n_frequencies]
    data_mono_batch = stft_mono_batch.transpose((0, 2, 1))
    data_music_batch = stft_music_batch.transpose((0, 2, 1))
    data_voice_batch = stft_voice_batch.transpose((0, 2, 1))
    return data_mono_batch, data_music_batch, data_voice_batch
    #调用 get_next_batch()
    data_mono_batch, data_music_batch, data_voice_batch = get_next_batch(
stfts_mono = stfts_mono_train_cut, stfts_music = stfts_music_train_cut,
stfts_voice = stfts_voice_train_cut,batch_size = batch_size,
sample_frames = sample_frames)
#获取频率值
x_mixed_src, _ = separate_magnitude_phase(data = data_mono_batch)
y_music_src, _ = separate_magnitude_phase(data = data_music_batch)
y_voice_src, _ = separate_magnitude_phase(data = data_voice_batch)
```

5. 设定相关路径与参数

```
#可以通过命令设置的参数
#dataset_dir:数据集路径
#model_dir:模型保存的文件夹
#model_filename:模型保存的文件名
#dataset_sr:数据集音频文件的采样率
#learning_rate:学习率
#batch_size:小批量训练数据的长度
#sample_frames:每次训练获取多少帧数据
#iterations:训练迭代次数
#dropout_rate:丢弃率
def parse_arguments(argv):
    parser = argparse.ArgumentParser()
    parser.add_argument('--dataset_train_dir', type=str, help='数据集训练数据路径',
default='./dataset/MIR-1K/Wavfile')
    parser.add_argument('--dataset_validate_dir', type=str, help='数据集验证数据路径',
default='./dataset/MIR-1K/UndividedWavfile')
    parser.add_argument('--model_dir', type=str, help='模型保存的文件夹', default=
'model')
    parser.add_argument('--model_filename', type=str, help='模型保存的文件名', default=
'svmrnn.ckpt')
    parser.add_argument('--dataset_sr', type=int, help='数据集音频文件的采样率', default
=16000)
    parser.add_argument('--learning_rate', type=float, help='学习率', default=0.0001)
    parser.add_argument('--batch_size', type=int, help='小批量训练数据的长度', default=64)
    parser.add_argument('--sample_frames', type=int, help='每次训练获取多少帧数据',
default=10)
    parser.add_argument('--iterations', type=int, help='训练迭代次数', default=30000)
    parser.add_argument('--dropout_rate', type=float, help='dropout率', default=0.95)
    return parser.parse_args(argv)
```

5.3.3　模型构建

将数据加载进模型之后，需要定义结构、优化模型和模型实现。

1. 定义结构

定义的架构为1024层循环神经网络，每层RNN的隐藏神经元个数从1～1024递增。最后是一个输入/输出的全连接层。在每层RNN之后，引入进行丢弃正则化，消除模型的过拟合问题。

模型初始化步骤：按顺序建立1024层RNN，每层拥有从1～1024递增的RNN神经元个数。原始混合数据通过RNN以及丢弃正则化之后，由ReLU函数激活，并利用全连接层输出音频特征值为513的纯伴奏和纯人声的数据。

为了约束输出y_music_src，y_voice_src的大小，即使输出的纯伴奏数据y_music_src

和纯人声数据 y_voice_src 与输入的混合数据 x_mixed_src 大小相同,输出需要进行以下变换:

$$y_m = \frac{d_m}{d_m + d_v + \alpha} \times x_m$$

$$y_v = \frac{d_v}{d_m + d_v + \alpha} \times x_m$$

$$y_m + y_v = \frac{d_m + d_v}{d_m + d_v + \alpha} \times x_m \approx x_m$$

其中,y_m 是 y_music_src,纯伴奏数据;y_v 是 y_voice_src,纯人声数据;d_m 是 y_dense_music_src,全连接层输出的纯伴奏数据;d_v 是 y_dense_voice_src,全连接层输出的人声数据;x_m 是 x_mixed_src,人声和伴奏的混合数据。

在分母上添加一个足够小的数 α,防止分母为 0。

```
#保存传入的参数
    self.num_features = num_features
    self.num_rnn_layer = len(num_hidden_units)
    self.num_hidden_units = num_hidden_units
    #设置变量
    #训练步数
    self.g_step = tf.Variable(0, dtype=tf.int32, name='g_step')
    #设置占位符
    #学习率
    self.learning_rate = tf.placeholder(tf.float32, shape=[], name='learning_rate')
    #混合了伴奏和人声的数据
    self.x_mixed_src = tf.placeholder(tf.float32, shape=[None, None, num_features], name=
'x_mixed_src')
    #伴奏数据
    self.y_music_src = tf.placeholder(tf.float32, shape=[None, None, num_features], name=
'y_music_src')
    #人声数据
    self.y_voice_src = tf.placeholder(tf.float32, shape=[None, None, num_features], name=
'y_voice_src')
    #保持丢弃,用于 RNN 网络
    self.dropout_rate = tf.placeholder(tf.float32)
    #初始化神经网络
    self.y_pred_music_src, self.y_pred_voice_src = self.network_init(
    #创建会话
    self.sess = tf.Session()
    #构建神经网络
def network_init(self):
    rnn_layer = []
    #根据 num_hidden_units 的长度来决定创建几层 RNN,每个 RNN 长度为 size
    for size in self.num_hidden_units:
    #使用 LSTM 保证大数据集情况下的模型准确度
```

```
    #加上丢弃,防止过拟合
        layer_cell = tf.nn.rnn_cell.LSTMCell(size)
        layer_cell = tf.contrib.rnn.DropoutWrapper(layer_cell, input_keep_prob = self.
dropout_rate)
        rnn_layer.append(layer_cell)
#创建多层 RNN
#为保证训练时考虑音频的前后时间关系,使用双向 RNN
    multi_rnn_cell = tf.nn.rnn_cell.MultiRNNCell(rnn_layer)
    outputs, state = tf.nn.bidirectional_dynamic_rnn(cell_fw = multi_rnn_cell, cell_bw =
multi_rnn_cell,
    inputs = self.x_mixed_src, dtype = tf.float32)
    out = tf.concat(outputs, 2)
#全连接层
#采用 ReLU 激活
    y_dense_music_src = tf.layers.dense(
        inputs = out,
        units = self.num_features,
        activation = tf.nn.relu,
        name = 'y_dense_music_src')
    y_dense_voice_src = tf.layers.dense(
        inputs = out,
        units = self.num_features,
        activation = tf.nn.relu,
        name = 'y_dense_voice_src')
    y_music_src = y_dense_music_src / (y_dense_music_src + y_dense_voice_src + np.finfo
(float).eps) * self.x_mixed_src
    y_voice_src = y_dense_voice_src / (y_dense_music_src + y_dense_voice_src + np.finfo
(float).eps) * self.x_mixed_src
    return y_music_src, y_voice_src
```

2. 优化模型

本项目使用的损失函数基于 reduce_mean(),计算输出的纯伴奏数据 y_music_src 及纯人声数据 y_voice_src 的方差。模型优化器选取 Adam 优化器,优化模型参数。

```
#损失函数
    def loss_init(self):
        with tf.variable_scope('loss') as scope:
            #求方差(reduce_mean 方法)
            loss = tf.reduce_mean(
                tf.square(self.y_music_src - self.y_pred_music_src)
                + tf.square(self.y_voice_src - self.y_pred_voice_src), name = 'loss')
        return loss
    #优化器
    #采取常用的 Adam 优化器
    def optimizer_init(self):
        optimizer = tf.train.AdamOptimizer(learning_rate = self.learning_rate).minimize
```

```
(self.loss)
        return optimizer
```

3. 模型实现

构建好模型结构、损失函数及优化器之后,定义训练、测试及保存的相关函数,便于使用。

```
#保存模型
    def save(self, directory, filename, global_step):
    #如果目录不存在,则创建
    if not os.path.exists(directory):
        os.makedirs(directory)
    self.saver.save(self.sess, os.path.join(directory, filename), global_step = global_step)
    return os.path.join(directory, filename)
    #加载模型,如果没有,则初始化所有变量
def load(self, file_dir):
    #初始化变量
    self.sess.run(tf.global_variables_initializer())
    #如果没有模型,重新初始化
    kpt = tf.train.latest_checkpoint(file_dir)
    print("kpt:", kpt)
    startepo = 0
    if kpt != None:
        self.saver.restore(self.sess, kpt)
        ind = kpt.find("-")
        startepo = int(kpt[ind + 1:])
    return startepo
    #开始训练
def train(self, x_mixed_src, y_music_src, y_voice_src, learning_rate, dropout_rate):
    #已经训练的步数
    #step = self.sess.run(self.g_step)
    _, train_loss = self.sess.run([self.optimizer, self.loss],
    feed_dict = {self.x_mixed_src: x_mixed_src, self.y_music_src: y_music_src, self.y_voice
_src: y_voice_src,
    self.learning_rate:learning_rate,self.dropout_rate: dropout_rate})
    return train_loss
#验证
def validate(self, x_mixed_src,y_music_src,y_voice_src, dropout_rate):
y_music_src_pred, y_voice_src_pred, validate_loss = self.sess.run([self.y_pred_music_src,
self.y_pred_voice_src, self.loss],
    feed_dict = {self.x_mixed_src: x_mixed_src, self.y_music_src: y_music_src, self.y_voice
_src: y_voice_src, self.dropout_rate: dropout_rate})
    return y_music_src_pred, y_voice_src_pred, validate_loss
#测试
def test(self, x_mixed_src, dropout_rate):
y_music_src_pred, y_voice_src_pred = self.sess.run([self.y_pred_music_src, self.y_pred_
```

```
voice_src],
feed_dict = {self.x_mixed_src: x_mixed_src, self.dropout_rate: dropout_rate})
return y_music_src_pred, y_voice_src_pred
```

5.3.4　模型训练及保存

本项目使用训练集和测试集拟合,具体包括模型训练及模型保存。

1. 模型训练

模型训练具体操作如下:

```
#初始化模型
    model = SVMRNN(num_features = n_fft // 2 + 1, num_hidden_units = num_hidden_units)
    #加载模型
    #如果没有模型,则初始化所有变量
    startepo = model.load(file_dir = model_dir)
    print('startepo:' + str(startepo))
    #开始训练
    #index 是切割训练集位置的标识符
    index = 0
    for i in (range(iterations)):
    #从模型中断处开始训练
        if i < startepo:
            continue
        wavs_mono_train_cut = list()
        wavs_music_train_cut = list()
        wavs_voice_train_cut = list()
    #从训练集中随机选取 64 个音频数据
        for seed in range(batch_size):
            index = np.random.randint(0,len(wavs_mono_train))
            wavs_mono_train_cut.append(wavs_mono_train[index])
            wavs_music_train_cut.append(wavs_music_train[index])
            wavs_voice_train_cut.append(wavs_voice_train[index])
        #短时傅里叶变换,将选取的音频数据转到频域
        stfts_mono_train_cut, stfts_music_train_cut, stfts_voice_train_cut = wavs_to_specs(
        wavs_mono = wavs_mono_train_cut, wavs_music = wavs_music_train_cut, wavs_voice =
wavs_voice_train_cut,
        n_fft = n_fft, hop_length = hop_length)
        #获取下一批数据
        data_mono_batch, data_music_batch, data_voice_batch = get_next_batch( stfts_mono =
stfts_mono_train_cut, stfts_music = stfts_music_train_cut, stfts_voice = stfts_voice_train_cut,
        batch_size = batch_size, sample_frames = sample_frames)
        #获取频率值
        x_mixed_src, _ = separate_magnitude_phase(data = data_mono_batch)
        y_music_src, _ = separate_magnitude_phase(data = data_music_batch)
        y_voice_src, _ = separate_magnitude_phase(data = data_voice_batch)
```

```
        #送入神经网络,开始训练
        train_loss = model.train(x_mixed_src = x_mixed_src, y_music_src = y_music_src, y_
voice_src = y_voice_src,learning_rate = learning_rate, dropout_rate = dropout_rate)
#每 10 步输出一次训练结果的损失值
        if i % 10 == 0:
            print('Step: %d Train Loss: %f' %(i, train_loss))
#每 200 步输出一次测试结果
        if i % 200 == 0:
            print('================================================')
            data_mono_batch, data_music_batch, data_voice_batch = get_next_batch(stfts_
mono = stfts_mono_valid, stfts_music = stfts_music_valid,stfts_voice = stfts_voice_valid,
batch_size = batch_size, sample_frames = sample_frames)
        x_mixed_src, _ = separate_magnitude_phase(data = data_mono_batch)
        y_music_src, _ = separate_magnitude_phase(data = data_music_batch)
        y_voice_src, _ = separate_magnitude_phase(data = data_voice_batch)
        y_music_src_pred, y_voice_src_pred, validate_loss = model.validate(x_mixed_src = x
_mixed_src,
        y_music_src = y_music_src, y_voice_src = y_voice_src, dropout_rate = dropout_rate)
            print('Step: %d Validation Loss: %f' %(i, validate_loss))
            print('================================================')
```

batch_size 是在一次前向/后向传播过程用到的训练样例数量,训练时随机选取 64 个数据并开始训练,总共训练 1110 个数据,迭代 30 000 步,如图 5-5 所示。

```
Step: 3600 Validation Loss: 1.106188
================================================
Step: 3610 Train Loss: 0.808172
Step: 3620 Train Loss: 1.425123
Step: 3630 Train Loss: 0.829051
Step: 3640 Train Loss: 1.110434
Step: 3650 Train Loss: 1.124288
Step: 3660 Train Loss: 0.998556
Step: 3670 Train Loss: 0.996172
Step: 3680 Train Loss: 1.189133
Step: 3690 Train Loss: 1.257032
Step: 3700 Train Loss: 1.298231
Step: 3710 Train Loss: 1.067620
Step: 3720 Train Loss: 0.947821
Step: 3730 Train Loss: 0.974504
Step: 3740 Train Loss: 0.957819
Step: 3750 Train Loss: 0.742355
Step: 3760 Train Loss: 0.977313
Step: 3770 Train Loss: 0.933429
```

图 5-5　训练结果

2. 模型保存

模型保存有两种作用：一是为了在训练过程中出现意外而中断时,能够在上次保存的模型处开始；二是为了在应用中直接使用训练好的模型。

```
#每 200 步保存一次模型
if i % 200 == 0:
  model.save(directory = model_dir, filename = model_filename, global_step = i)
```

5.3.5　模型测试

采用 Python 自带的 Tkinter 库进行 GUI 设计，GUI 实现如下功能。

1. 批量选取歌曲和保存路径

定义 addfile()和 choose_save_path()两个函数，使用 Tkinter 中 filedialog 模块打开系统路径。

```
def addfile():
    #定义全局变量 music_path,用于添加音频文件
    global music_path
    paths = tk.filedialog.askopenfilenames(title='选择要分离的歌曲')
    #保存选择的歌曲
    #遍历添加
    for path in paths:
        music_path.append(path)
    label_info['text'] = '\n'.join(music_path)
    #选择分离结束后保存的文件夹
def choose_save_path():
    global save_path
    save_path = tk.filedialog.askdirectory(title='选择保存文件夹')
    save_info['text'] = save_path
```

2. 模型导入及调用

定义调用模型的函数 separate()。该函数完成如下功能：

(1) 将带转化的文件进行列表保存、数据分割、短时傅里叶变换，完成数据的预处理。

```
#加载音频文件
wavs_mono = list()
for filename in music_path:
      wav_mono, _ = librosa.load(filename, sr=dataset_sr, mono=True)
      wavs_mono.append(wav_mono)
#短时傅里叶变换的 fft 点数
#默认情况下,窗口长度 = fft 点数
n_fft = 1024
#冗余度
hop_length = n_fft // 4
#将音频数据转换到频域
stfts_mono = list()
for wav_mono in wavs_mono:
stft_mono = librosa.stft(wav_mono,n_fft=n_fft,hop_length=hop_length)
stfts_mono.append(stft_mono.transpose())
```

(2) 数据预处理后，调用训练好的模型，进行人声和伴奏的分离。

```
#初始化神经网络
```

```
model = SVMRNN(num_features = n_fft // 2 + 1, num_hidden_units = num_hidden_units)
#导入模型
model.load(file_dir = model_dir)
for wav_filename, wav_mono, stft_mono in zip(music_path, wavs_mono, stfts_mono):
wav_filename_base = os.path.basename(wav_filename)
#单声道音频文件
wav_mono_filename = wav_filename_base.split('.')[0] + '_mono.wav'
#分离后的纯伴奏音频文件
wav_music_filename = wav_filename_base.split('.')[0] + '_music.wav'
#分离后的纯人声音频文件
wav_voice_filename = wav_filename_base.split('.')[0] + '_voice.wav'
  #要保存文件的相对路径
    wav_mono_filepath = os.path.join(save_path, wav_mono_filename)
    wav_music_hat_filepath = os.path.join(save_path,wav_music_filename)
    wav_voice_hat_filepath = os.path.join(save_path,wav_voice_filename)
        print('Processing %s ...' % wav_filename_base)
        stft_mono_magnitude, stft_mono_phase = separate_magnitude_phase(data = stft_mono)
        stft_mono_magnitude = np.array([stft_mono_magnitude])
        y_music_pred, y_voice_pred = model.test(x_mixed_src = stft_mono_magnitude, dropout_
rate = dropout_rate)
```

(3) 将处理好的频率文件和原本的相位信息相加,进行傅里叶逆变换。

```
#根据振幅和相位,得到复数
#信号 s(t)乘上 e^(j*phases)表示信号 s(t)移动相位 phases
def combine_magnitude_phase(magnitudes, phases):
    return magnitudes * np.exp(1.j * phases)
```

(4) 将时域文件写成.wav 格式的歌曲保存。

```
  #保存数据,使用 librosa.output.write_wav()函数,将文件保存成.wav 格式歌曲文件
librosa.output.write_wav(wav_mono_filepath, wav_mono, dataset_sr)
librosa.output.write_wav(wav_music_hat_filepath,y_music_hat,dataset_sr)
librosa.output.write_wav(wav_voice_hat_filepath,y_voice_hat,dataset_sr)
  #检测在保存文件夹中是否生成了伴奏文件,若存在则自动打开该文件夹
  if os.path.exists(wav_music_hat_filepath):
      os.startfile(save_path)
      remind_window.destroy()
```

3. GUI 代码

GUI 相关代码如下:

```
from tkinter import Tk, filedialog
import tkinter as tk
import librosa
import os
import numpy as np
from NewModel import SVMRNN
```

```
from NewUtils import separate_magnitude_phase, combine_magnitude_phase
music_path = []
save_path = str()
wav_music_hat_filepath = str()
def addfile():
    #定义全局变量 music_path,用于添加音频文件
    global music_path
    paths = tk.filedialog.askopenfilenames(title='选择要分离的歌曲')
    #保存选择的歌曲
    #遍历添加
    for path in paths:
        music_path.append(path)
    label_info['text'] = '\n'.join(music_path)
    #选择分离结束后保存的文件夹
def choose_save_path():
    global save_path
    save_path = tk.filedialog.askdirectory(title='选择保存文件夹')
    save_info['text'] = save_path
    #弹窗信息的定义
def pop_window():
    global wav_music_hat_filepath
    def separate():
        dataset_sr = 16000        #采样率
        model_dir = './model'     #模型保存文件夹
        dropout_rate = 0.95       #丢弃率
        #加载音频文件
        wavs_mono = list()
        for filename in music_path:
            wav_mono, _ = librosa.load(filename, sr=dataset_sr, mono=True)
            wavs_mono.append(wav_mono)
        #短时傅里叶变换的 fft 点数
        #默认情况下,窗口长度 = fft 点数
        n_fft = 1024
        #冗余度
        hop_length = n_fft // 4
        #用于创建 RNN 节点数
        num_hidden_units = [1024, 1024, 1024, 1024, 1024]
        #将音频数据转换到频域
        stfts_mono = list()
        for wav_mono in wavs_mono:
        stft_mono = librosa.stft(wav_mono,n_fft=n_fft,hop_length=hop_length)
        stfts_mono.append(stft_mono.transpose())
        #初始化神经网络
model = SVMRNN(num_features=n_fft//2+1,num_hidden_units=num_hidden_units)
        #导入模型
        model.load(file_dir=model_dir)
        for wav_filename, wav_mono, stft_mono in zip(music_path, wavs_mono, stfts_mono):
```

```
            wav_filename_base = os.path.basename(wav_filename)
            #单声道音频文件
            wav_mono_filename = wav_filename_base.split('.')[0] + '_mono.wav'
            #分离后的纯伴奏音频文件
              wav_music_filename = wav_filename_base.split('.')[0] + '_music.wav'
            #分离后的纯人声音频文件
            wav_voice_filename = wav_filename_base.split('.')[0] + '_voice.wav'
            #要保存文件的相对路径
            wav_mono_filepath = os.path.join(save_path, wav_mono_filename)
            wav_music_hat_filepath = os.path.join(save_path,wav_music_filename)
            wav_voice_hat_filepath = os.path.join(save_path,wav_voice_filename)
                print('Processing %s ...' % wav_filename_base)
                stft_mono_magnitude, stft_mono_phase = separate_magnitude_phase(data = stft_mono)
                stft_mono_magnitude = np.array([stft_mono_magnitude])
                 y_music_pred, y_voice_pred = model.test(x_mixed_src = stft_mono_magnitude,
dropout_rate = dropout_rate)
                #根据振幅和相位,转为复数,用于下面的逆短时傅里叶变换
                 y_music_stft_hat = combine_magnitude_phase(magnitudes = y_music_pred[0],
phases = stft_mono_phase)
                 y_voice_stft_hat = combine_magnitude_phase(magnitudes = y_voice_pred[0],
phases = stft_mono_phase)
                y_music_stft_hat = y_music_stft_hat.transpose()
                y_voice_stft_hat = y_voice_stft_hat.transpose()
                #通过逆短时傅里叶变换,将分离好的频域数据转换为音频,生成相对应的音频文件
                y_music_hat = librosa.istft(y_music_stft_hat, hop_length = hop_length)
                y_voice_hat = librosa.istft(y_voice_stft_hat, hop_length = hop_length)
                #保存数据
librosa.output.write_wav(wav_mono_filepath,wav_mono,dataset_sr)
                librosa.output.write_wav(wav_music_hat_filepath,y_music_hat, dataset_sr)
                librosa.output.write_wav(wav_voice_hat_filepath,y_voice_hat, dataset_sr)
                if os.path.exists(wav_music_hat_filepath):
                    os.startfile(save_path)
                    remind_window.destroy()
    remind_window = tk.Toplevel()
    remind_window.title('提示')
    remind_window.minsize(width = 400, height = 200)
    tk.Label(remind_window, text = '加载模型中,请勿关闭软件').place(x = 70, y = 60)
    tk.Button(remind_window, text = '我知道了', font = ('Fangsong', 14), command = separate).
place(x = 150, y = 100)
root = Tk()
#窗口标题
root.title('歌曲人声分离')
#大小不可调整
root.resizable(0,0)
#创建背景图片
canvas = tk.Canvas(root, width = 800, height = 900, bd = 0, highlightthickness = 0)
imgpath = 'image/bg1.jpg'
```

```
img = Image.open(imgpath)
photo = ImageTk.PhotoImage(img)
#设置背景图片在窗口显示的偏移量
canvas.create_image(750, 400, image = photo)
canvas.pack()
#添加按钮
#添加按钮的图片
btn_add = tk.PhotoImage(file = 'image/btn_add.png')
btn_addfile = tk.Button(root, command = addfile, image = btn_add)
#将按钮摆放到窗口上
canvas.create_window(70,70, width = 84, height = 84, window = btn_addfile)
#功能说明文字
canvas.create_text(70,130, text = '添加文件', fill = 'white', font = ('Fangsong', '15', 'bold'))
#选择保存路径的按钮
pic_save = tk.PhotoImage(file = 'image/btn_save.png')
btn_save = tk.Button(root, command = choose_save_path, image = pic_save)
canvas.create_window(200, 70, width = 84, height = 84, window = btn_save)
canvas.create_text(200,130,text = '选择保存路径',fill = 'white', font = ('Fangsong', '15', 'bold'))
#运行按钮
pic_run = tk.PhotoImage(file = 'image/btn_run.png')
btn_run = tk.Button(root, command = pop_window, image = pic_run)
canvas.create_window(330, 70, width = 84, height = 84, window = btn_run)
canvas.create_text(330, 130, text = '运行', fill = 'white', font = ('Fangsong', '15', 'bold'))
#显示待分离歌曲
canvas.create_text(70, 180, text = '待分离的歌曲', fill = 'white', font = ('Fangsong', '14', 'bold'))
label_info = tk.Label(root, bg = 'white', anchor = 'nw', justify = 'left')
canvas.create_window(210, 300, width = 400, height = 200, window = label_info)
#显示保存的路径
canvas.create_text(48, 430, text = '保存路径', fill = 'white', font = ('Fangsong', '14', 'bold'))
save_info = tk.Label(root, bg = 'white', anchor = 'nw', justify = 'left')
canvas.create_window(210, 470, width = 400, height = 50, window = save_info)
root.mainloop()
```

5.4　系统测试

本部分包括训练准确率、测试效果及模型应用。

5.4.1　训练准确率

训练迭代到靠后的步数，损失函数的值小于 0.5，这意味着这个预测模型训练比较成功。在整个迭代训练过程中，随着 epoch 的增加，模型损失函数的值在逐渐减小，并且在 17 000 步以后趋于稳定，如图 5-6 所示。

```
================================================
Step: 17200 Validation Loss: 0.420042
================================================
Step: 17210 Train Loss: 0.507203
Step: 17220 Train Loss: 0.502529
Step: 17230 Train Loss: 0.541619
Step: 17240 Train Loss: 0.580664
Step: 17250 Train Loss: 0.450910
Step: 17260 Train Loss: 0.628350
Step: 17270 Train Loss: 0.479427
Step: 17280 Train Loss: 0.497587
Step: 17290 Train Loss: 0.542923
Step: 17300 Train Loss: 0.432395
Step: 17310 Train Loss: 0.469025
Step: 17320 Train Loss: 0.444656
Step: 17330 Train Loss: 0.449266
```

图 5-6　训练准确率

5.4.2　测试效果

将数据代入模型进行测试，并将分离得到的纯伴奏和纯人声波形经过对比以及人耳辨别，得到验证：模型可以实现歌曲伴奏和人声分离。如图 5-7 和图 5-8 所示。

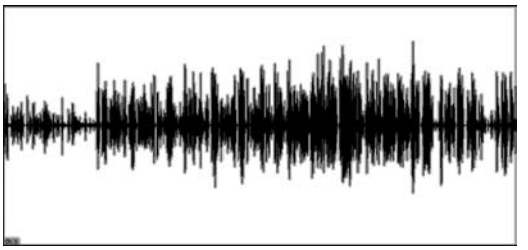

图 5-7　分离的纯人声波形

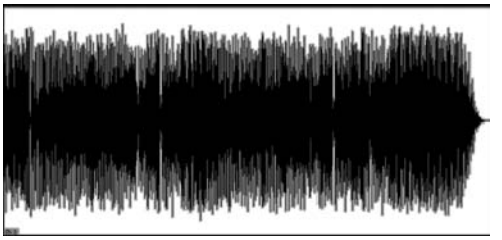

图 5-8　分离的纯伴奏波形

5.4.3　模型应用

使用说明包括以下 4 部分。

(1) 打开 gui.py 文件，界面如图 5-9 所示。

(2) 单击“添加文件”按钮，选择要获得伴奏的歌曲，在“待分离的歌曲”中显示添加歌曲的路径和名称，如图 5-10 所示。

图 5-9　主界面

图 5-10　添加歌曲

(3) 选择保存路径,如图 5-11 所示。

图 5-11 保存路径

(4) 单击“运行”按钮后会弹出提示框,确认后程序开始运行,运行完毕后将自动打开保存文件夹,如图 5-12 所示。

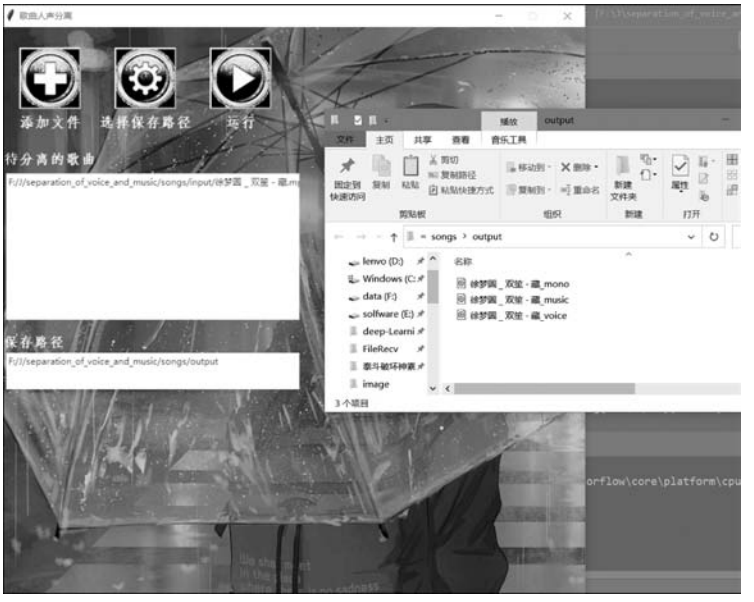

图 5-12 运行结果