基于梅尔频谱特征和长短期记忆网络的语音转换系统

温泽林（学号：202200810234）

2024 年 5 月

摘要

语音转换技术可以将说话者的语音特征转化为另一个说话者，同时保证语音内容不变。在传统的语音转换方法中，基于统计建模的方法虽然在一定程度上取得成功，但在合成语音的自然性和清晰度等方面依然存在局限性，且难以对大量语音数据进行处理。近年来，语音信号处理技术的发展和长短期记忆网络的出现为语音转换研究提供了新的契机。本文结合音频信号的梅尔频谱特征和长短期记忆网络等技术，构建了一个的高效语音转换系统，具有高速度、高准确率、易迁移等优点。

1 引言

1.1 研究背景

语音转换技术在保证言语内容不变的前提下，将一个人的声音转变为另一个说话者。语音转换的过程通常包括三部分，即特征提取、特征映射和语音重建，涉及语音分析、频率转换、语音特征编码、数学建模等多种技术。[3] 语音转换采用的数据分为平行语料与非平行语料两种，主要区别为是否具有相同语义内容的成对语音数据。本文所述系统采用的是平行语料。相较于非平行语料而言，平行语料有着转换速度快、对数据规模要求小等优点。[11]

语音转换技术中最有挑战性的研究问题在于语音属性的操纵。[5] 20 世纪 70 年代，数字信号处理技术的飞速发展极大地促进了人类对音频特征的有效控制。语音转换的早期研究大多针对于平行语料，采用统计模型对语音频谱进行映射，如鹿野等人提出的模糊向量量化法 [2]、赫兰德等人提出的

动态时间扭曲方法 [1]、户田智基等人提出的高斯混合模型方法 [6] 等。这些方法通常基于统计理论和数学建模方法，达到了相对优秀的效果，但方法理论过于复杂、对不同数据的通用性也存在局限性，这些因素限制了这些方法在日常生活中的应用。

近年来以来，随着自然语言处理、随机过程理论和计算机硬件的不断发展，基于深度学习的语音转换技术取得了显著进展。[3] 这种方法依然沿用特征提取、特征映射和语音重建的流程，但可以通过端到端训练的手段，直接让计算机软件自主提取有利于语音转换的特征，从而减少对人工干预的需求。这不仅提高了系统的灵活性和适应性，还能更好地捕捉语音信号中的复杂模式和细微差异。[7] 此外，端到端训练还使得模型能够更高效地处理多样化的输入数据，提高了语音转换的鲁棒性和准确性。

1.2 理论基础

1.2.1 短时傅里叶变换

短时傅里叶变换（STFT）是一种针对时变、非平稳信号的联合时频分析方法，能将一维的声波信号变换成适于长短期记忆网络处理的二维矩阵，一种包含时域和频域信息的特征谱。[9]

短时傅里叶变换的基本思想是采用固定长度的窗函数对时域信号进行截取，并对截取得到的信号进行傅里叶变换，得到以时刻 t 为中心的时间帧的局部频谱。窗函数在音频信号的整个时间轴上进行平移，便可以得到每一帧上局部频谱的集合，构成二维坐标系下的特征谱图。它的基本运算公式如下：

$$\mathrm{STFT}_f(t,\omega)=\int_{-\infty}^{\infty} f(t)g(t-\tau)\mathrm{e}^{-\mathrm{j}\omega t}\mathrm{d}t$$

其中 $f(t)$ 为时域信号，$g(t-\tau)$ 为中心位于 τ 时刻的窗函数。

为了缓解截取信号时产生的频谱泄露现象，本系统采用汉明窗（Hamming window）[4] 作为窗函数，其数学表达式为

$$W(m)=\begin{cases}0.54-0.46\cos\left(\dfrac{2\pi m}{M-1}\right) & 0\le m\le M-1\\ 0 & \text{otherwise}\end{cases}.$$

1.2.2 梅尔频谱

人耳对于频率的感知不是线性的，相比于高频而言，人类对低频更加敏感。梅尔频谱是更符合人耳的听觉特性的一种频域表示法，它在 1000Hz 以下呈线性分布，1000Hz 以上呈对数增长。[8] 就具体操作而言，我们对信号进行短时傅里叶变换得到线性频谱，再用一组梅尔滤波器处理线性频谱，即可得到梅尔频谱。下图是梅尔频谱的一个示例：

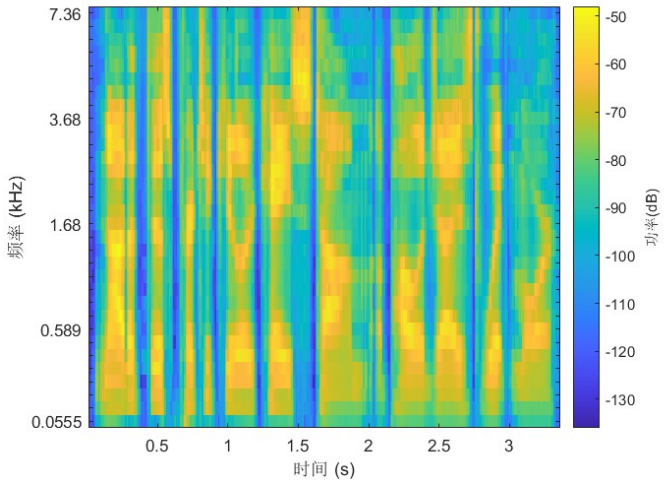

图 1: 梅尔频谱示例图

1.2.3 深度神经网络

深度学习是一类方法的统称通过模仿人类的神经元机制来刻画数据。它利用人工构建的深度神经网络结构学习数据的内在规律，通过组合低层特征形成高层的抽象特征。目前深度学习技术在自然语言处理、图像处理等领域取得成功，在许多任务上的效果超出其他模型。

1.2.4 循环神经网络

循环神经网络是一类特殊的深度神经网络，它将时序的概念引入到网络设计中，使其在时序数据（如音频信号）的分析中表现出更强的适应性。但传统的循环神经网络存在梯度消失、梯度爆炸和长期记忆能力不足等问题。[10] 长短期记忆网络（LSTM）是循环神经网络的一个变种，可以有效

的解决上述问题，使得循环神经网络能够真正地利用数据上下文之间的关联信息。

2 数据集简介

系统所使用的数据全部来自卡内基梅隆大学语言技术研究所构建的 CMU_ARCTIC 语音合成数据库。训练模型时，选取美式英语（bdl 男性）作为原音频，美式英语（slt 女性）作为目标音频。原音频与目标音频为平行语料，由 1132 组两人朗读相同内容的音频对构成。所有音频的采样率均为 16000Hz，音频长度在 1 秒到 6 秒不等。

3 系统架构

本文介绍的语音转换系统由四个模块组成：音频预处理模块、特征提取模块、特征映射模块和音频重建模块。其中，音频预处理模块对原语音信号进行预先处理，包括去噪、预加重等；特征提取模块从音频信号构建梅尔频谱图，从而提取出有用的、便于后续神经网络处理的特征，同时尽可能减少语音信息的损失；特征映射模块以长短期神经网络和全连接网络为骨干构建了一个序列—序列模型，将原语音音频的特征映射到目标语音特征；音频重建模块则将映射后的特征还原为音频信号，作为最终的转换结果。

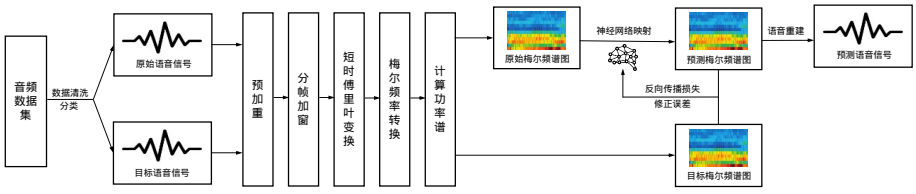

图 2: 网络架构图

3.1 音频预处理模块

音频预处理模块主要为预加重操作。人类的发声器官在向外辐射声波的时候，空气作为语音信号的负载一方面传播着能量，另一方面也在损耗能量。具体地说，介质作为声能量的载体，在声源尺寸一定的情况下，频率越高，介质对声能量的损耗越严重。为了弥补高频分量的能量损失，需要对语音信号进行滤波，这一操作被称为预加重。常见的预加重滤波器为高通滤波器，其传递函数为

$$H(z) = 1 - az^{-1}$$

本系统中选取的预加重参数 α 为 0.97。

如图所示为一个声波信号经过预加重操作前后的梅尔频谱图。

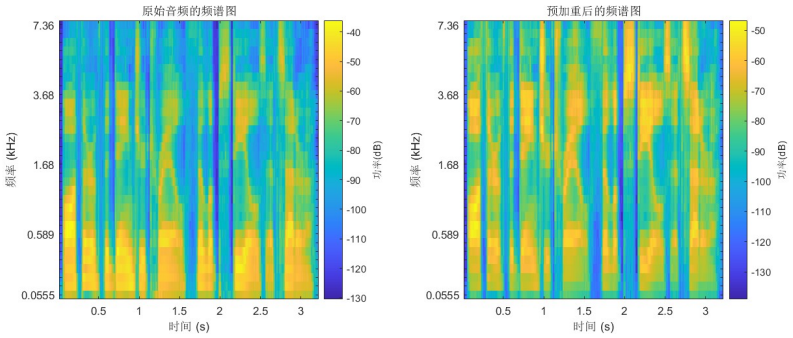

图 3: 预加重前后的梅尔频谱图，左为预加重前，右为预加重后

可以看出，预加重操作后，原始声波的高频信号得到明显增强，而低频信号几乎保持不变。这说明预加重操作有效弥补了高频分量在传播过程中的能量损失。

3.2 特征提取模块

特征提取模块接受经过预处理的音频，对其进行分帧加窗、短时傅里叶变换、梅尔滤波等步骤，构建出输入音频的梅尔频谱图作为信号。帧长设定为 0.03 秒（即 480 个采样点）。为了保证语音特征的连续性，在分帧的过程中，设置两帧之间有 0.02 秒（320 个采样点）的重叠。加窗时，采用汉明窗作为窗口函数。

3.3 特征映射模块

特征映射模块采用长短期记忆网络和全连接神经网络为结构骨架，构建了一个端到端的序列—序列模型（LSTM-Seq2seq）。模型架构如下图所示：

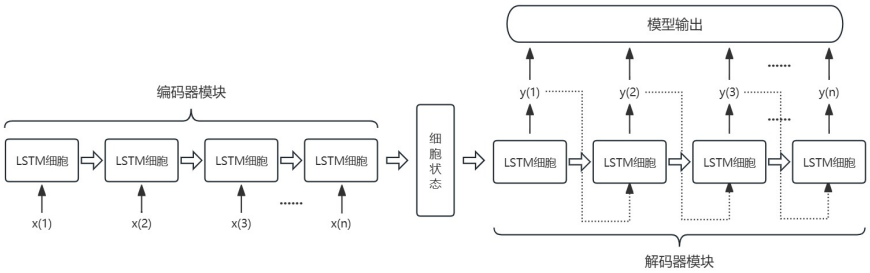

图 4: LSTM-Seq2seq 模型架构

音频信号首先在 LSTM 编码器中进行编码和进一步特征提取，此时的 LSTM 细胞状态反映了原音频信号的特征；将细胞状态经过多层全连接网络进行映射，映射后的细胞状态即为目标音频信号对应的特征；将细胞状态经过 LSTM 解码器解码，即可得到目标音频的梅尔频谱图。

音频信号在 LSTM 网络内部的传播过程可以用数学公式刻画。将音频信号在第 t 个时间帧上的梅尔频谱特征向量记为 x_t，则第 t 帧信号在长短期记忆网络内部的前向传播方程可以表示为：

$$
\begin{aligned}
i_t &= \sigma(W_i \odot [h_{t-1}, x_t] + b_i) \\
f_t &= \sigma(W_f \odot [h_{t-1}, x_t] + b_f) \\
g_t &= \tanh(W_C \odot [h_{t-1}, x_t] + b_C) \\
o_t &= \sigma(W_o \odot [h_{t-1}, x_t] + b_o) \\
c_t &= f_t * c_{t-1} + i_t * g_t \\
h_t &= o_t * \tanh(c_t)
\end{aligned}
$$

其中 i_t 为输入门，f_t 为遗忘门，o_t 为输出门，c_t 为记忆状态，h_t 为隐藏状态。可以直观地理解为，第 t 层长短期记忆细胞接收前 $t-1$ 帧音频的构建的记忆状态和第 t 帧音频作为输入，在一系列门控单元的运算过程中，

计算出当前帧音频对应的目标语音，同时更新细胞记忆状态，供下一帧音频参考。

3.4 音频重建模块

音频重建模块采用 Griffin-Lim 信号估计算法将映射后的梅尔频谱重建为时域信号，使得估计得到的信号傅里叶变换的幅度值与原始信号傅里叶变换的幅度值的平方误差达到最小。

$$x^i(n) = \frac{\sum_{n=-\infty}^{\infty} \omega(ms-n) \int_{-\pi}^{\pi} X^{(i)}(m,n) \mathrm{e}^{\mathrm{j}\omega n} \mathrm{d}\omega}{2\pi \sum_{i=-\infty}^{\infty} \omega^2(ms-n)}$$

通过迭代重构信号的相位信息和已知的幅度信息，得到语音信号的估算值。比较估算的语音信号傅里叶变换的幅度值与原始信号幅度值的差值，其中，$\omega(ms-n)$ 为一个窗序列，s 表示窗移的长度，$x_i(n)$ 是第 i 次迭代后重构的语音信号。当两者的差值达到一个比较小的值后，则停止迭代，就认为第 i 次迭代后所获得的语音信号是从信号傅里叶变换幅度值重构的原始语音信号。[12]

4 评估指标

我们将均方误差指标（MSE）作为网络训练的损失函数，并用其评估模型。在本问题中，MSE 的计算公式如下：

$$\mathrm{MSE} = \frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} (y_{ij} - \hat{y}_{ij})^2$$

其中，y_{ij} 表示目标音频的梅尔频谱功率，$\hat{y}_{ij}$ 表示预测音频的梅尔频谱功率。MSE 的值越小，意味着系统合成的音频与目标音频越相似、模型精确度越高。

5 实验环境

系统选用 MATLAB 2024a 和 PyTorch 2.2.0 作为软件平台，音频预处理模块和特征提取模块构建在 MATLAB 上，特征映射模块和音频重建模块使用 PyTorch 深度学习框架构建。硬件环境选用 Intel Xeon Gold 6130 处理器和 NVIDIA Tesla V100 Volta 图形处理器。

6 实验结果

使用 Adam 优化器与 10^{-7} 的学习率对训练数据进行了 10 轮训练。模型损失函数曲线如下图所示。

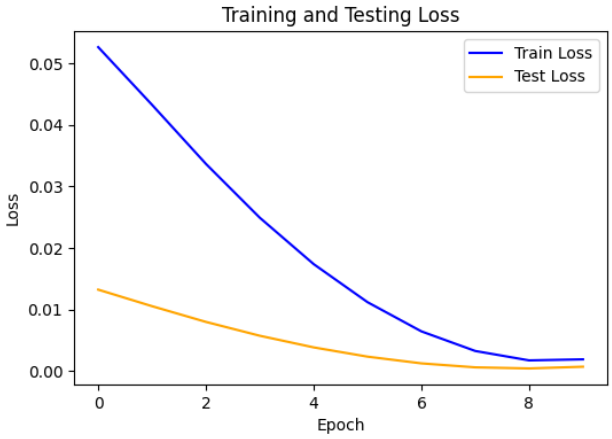

图 5: 损失函数曲线

经过训练数据的训练，损失函数的数值下降到了较低水平，这意味着模型可以较好地拟合特征映射过程。同时还可以注意到，模型在训练集与测试集上都有较好的效果，这说明模型具有良好的泛化能力，可以充分适应不同种类的语音输入。

随机选取测试集中的一组数据，其在实验过程中生成的梅尔频谱如下：

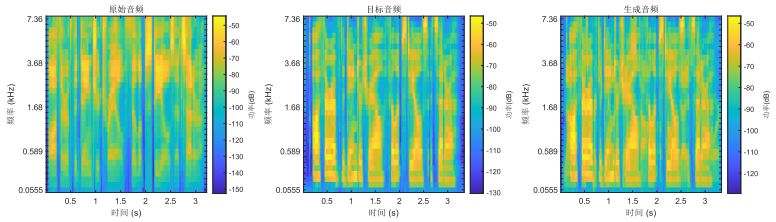

图 6: 梅尔频谱对比，从左至右分别为原始音频、目标音频、生成音频

可以看出，系统生成的音频与目标音频具有较好的相合性，可以较好地反映目标音频的频域特性。

7 讨论

本文提出的语音转换系统可以同时处理大量音频数据，具有转换效率高、准确性高、抗噪能力强等优点。在输入语音数据后，首先对音频数据进行去噪和预加重处理，有利于模型更准确地提取出音频特征，增强系统鲁棒性。特征提取模块构建原始语音信号的梅尔频谱图作为特征，可以将信号变换为便于计算机处理的形式，同时最大限度地保留原始信号特征。特征映射模块采用的长短期记忆网络可以有效捕捉音频信号中的长期依赖关系，有利于解决传统循环神经网络在处理长序列时容易出现的梯度消失问题。此外，端到端的训练过程还有利于适应不同种类的音频数据，便于系统迅速迁移到其他数据集上。

文章所述系统使用的所有代码可参考附录部分。

参考文献

[1] Elina Helander et al. “On the impact of alignment on voice conversion performance”. In: Sept. 2008, pp. 1453–1456. DOI: 10.21437/Interspeech.2008-419.

[2] Kiyohiro Shikano, Satoshi Nakamura, and Masanobu Abe. “Speaker adaptation and voice conversion by codebook mapping”. English. In: *Proceedings - IEEE International Symposium on Circuits and Systems* 1 (1991). 1991 IEEE International Symposium on Circuits and Sys-

tems Part 1 (of 5) ; Conference date: 11-06-1991 Through 14-06-1991, pp. 594–597. ISSN: 0271-4310.

[3] Berrak Sisman et al. “An Overview of Voice Conversion and Its Challenges: From Statistical Modeling to Deep Learning”. In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 29 (2021), pp. 132–157. DOI: 10.1109/TASLP.2020.3038524.

[4] J Sophocles. *Introduction to signal processing*. Prentice-Hall, Inc, 1996.

[5] JOHN Q. STEWART. “An Electrical Analogue of the Vocal Organs”. In: *Nature* 110.2757 (1922), pp. 311–312. ISSN: 1476-4687. DOI: 10.1038/110311a0. URL: https://doi.org/10.1038/110311a0.

[6] Tomoki Toda, Alan W. Black, and Keiichi Tokuda. “Voice Conversion Based on Maximum-Likelihood Estimation of Spectral Parameter Trajectory”. In: *IEEE Transactions on Audio, Speech, and Language Processing* 15.8 (2007), pp. 2222–2235. DOI: 10.1109/TASL.2007.907344.

[7] Tomasz Walczyna and Zbigniew Piotrowski. “Overview of Voice Conversion Methods Based on Deep Learning”. In: *Applied Sciences* 13.5 (2023). ISSN: 2076-3417. DOI: 10.3390/app13053100. URL: https://www.mdpi.com/2076-3417/13/5/3100.

[8] 何东升. “基于深度学习的中文语音合成技术研究与实现”. MA thesis. 东南大学, 2022.

[9] 李恒 et al. “基于短时傅里叶变换和卷积神经网络的轴承故障诊断方法”. In: *振动与冲击* 37.19 (2018), pp. 124–131. ISSN: 1000-3835. DOI: 10.13465/j.cnki.jvs.2018.19.020.

[10] 王鑫 et al. “基于 LSTM 循环神经网络的故障时间序列预测”. In: *北京航空航天大学学报* 44.04 (2018), pp. 772–784. ISSN: 1001-5965. DOI: 10.13700/j.bh.1001-5965.2017.0285.

[11] 缪承志. “基于平行语料的语音情感分析与转换”. MA thesis. 北方工业大学, 2024.

[12] 邱泽宇, 屈丹, and 张连海. “基于 WaveNet 的端到端语音合成方法”. In: *计算机应用* 39.05 (2019), pp. 1325–1329. ISSN: 1001-9081.

附录　系统设计采用的代码

A　音频预处理模块（MATLAB）

A.1　读取音频数据

```
train_data = load_data('./data/train/source', './data/train/target');

function data = load_data(train_source, train_target)
    file_info = dir(train_source);
    data = struct('source', {}, 'target', {});
    cnt = 1;
    for i = 1:length(file_info)
        filename = file_info(i).name;
        if not(endsWith(filename, '.wav'))
            continue
        end
        [data(cnt).source, data(cnt).fs] = audioread(train_source + "/"
        ↪  + filename);
        data(cnt).target = audioread(train_target + "/" + filename);
        cnt = cnt + 1;
    end
end
```

A.2　预加重

```
for i = 1:length(train_data)
    train_data(i).source = pre_emphasis(train_data(i).source);
    train_data(i).target = pre_emphasis(train_data(i).target);
end

% 接受原始时域信号，返回预加重后的时域信号
function y = pre_emphasis(x, alpha)
    if nargin < 2       % alpha 值默认为 0.97
        alpha = 0.97;
    end
    y = filter([1, -alpha], 1, x);
end
```

A.3 分帧加窗

```
% x: 时域信号
% fs: 采样率
% frame_len: 帧长度
% frame_step: 相邻两帧之间起始位置的间隔
% win_func: 窗函数
function frames = framesig(x, fs, frame_len, frame_step, win_func)
    frame_len = round(frame_len * fs / 1000);
    frame_step = round(frame_step * fs / 1000);
    audio_len = length(x);
    if frame_len >= audio_len
        n_frames = 1;                              % 帧数
    else
        n_frames = floor((audio_len - frame_len) / frame_step) + 1;
    end
    win = win_func(n_frames)';                     % 窗函数的值
    frames = zeros(frame_len, n_frames);
    for i = 1:n_frames
        start_idx = (i-1) * frame_step + 1;
        end_idx = start_idx + frame_len - 1;
        frames(:, i) = x(start_idx : end_idx);  % 分帧并与窗函数相乘
    end
    frames = frames .* win;
end
```

B 特征提取模块（MATLAB）

```
for i = 1:length(train_data)
    train_data(i).source_graph = melSpectrogram(train_data(i).source,
    ↪  train_data(i).fs);
    train_data(i).target_graph = melSpectrogram(train_data(i).target,
    ↪  train_data(i).fs);
end
save('train_data.mat', 'train_data');       % 将提取出的特征保存到文件
```

C　特征映射模块（Python）

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, TensorDataset
import scipy.io
import matplotlib.pyplot as plt

# 将音频特征加载进内存
train_data = scipy.io.loadmat('train_data.mat')
train_data = train_data['train_data']
source_graph = train_data['source_graph'][0]
target_graph = train_data['target_graph'][0]

source_graph = [i.T for i in source_graph]
target_graph = [i.T for i in target_graph]

# 为便于神经网络处理，首先将所有音频的长度全部变为 10 秒，不足 10 秒的在原音
↪  频后面添加静音片段
def pad_input(x):
    padded = [torch.tensor(i, dtype=torch.float) for i in x]
    padded = pad_sequence(padded, batch_first=True)
    padded = F.pad(padded, (0, 0, 0, 1024 - padded.shape[1]),
    ↪  'constant', 0)
    return padded

n_data = len(source_graph)              # 数据长度
n_train = round(n_data * 0.8)           # 训练集长度
n_test = n_data - n_train               # 测试集长度

source_graph = pad_input(source_graph)
target_graph = pad_input(target_graph)

train_source = source_graph[:n_train]
train_target = target_graph[:n_train]
test_source = source_graph[n_train:]
test_target = target_graph[n_train:]
```

```
INPUT_SIZE = 32              # 输入特征维度
HIDDEN_SIZE = 256            # 隐藏层维度
OUTPUT_SIZE = 32             # 输出特征维度

# 编码器定义
class Encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.in_features = INPUT_SIZE
        self.hidden_size = HIDDEN_SIZE
        self.encoder = nn.LSTM(input_size=INPUT_SIZE,
        ↪  hidden_size=HIDDEN_SIZE, dropout=0.5, num_layers=1)  #
        ↪  encoder

    def forward(self, enc_input: torch.Tensor):
        seq_len, batch_size, embedding_size = enc_input.size()
        h_0 = torch.rand(1, batch_size, self.hidden_size, device='cuda')
        c_0 = torch.rand(1, batch_size, self.hidden_size, device='cuda')
        encode_output, (encode_ht, decode_ht) = self.encoder(enc_input,
        ↪  (h_0, c_0))
        return encode_output, (encode_ht, decode_ht)

# 解码器定义
class Decoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.in_features = INPUT_SIZE
        self.hidden_size = HIDDEN_SIZE
        self.fc = nn.Linear(HIDDEN_SIZE, INPUT_SIZE)
        self.decoder = nn.LSTM(input_size=INPUT_SIZE,
        ↪  hidden_size=HIDDEN_SIZE, dropout=0.5, num_layers=1)  #
        ↪  encoder

    def forward(self, enc_output, dec_input):
        (h0, c0) = enc_output
        de_output, (_, _) = self.decoder(dec_input, (h0, c0))
        return de_output
```

```
# LSTM-Seq2seq 整体架构定义
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = Encoder()
        self.decoder = Decoder()
        self.in_features = INPUT_SIZE
        self.hidden_size = HIDDEN_SIZE
        self.fc = nn.Linear(HIDDEN_SIZE, INPUT_SIZE)

    def forward(self, enc_input):
        enc_input = enc_input.permute(1, 0, 2)
        # [seq_len, batch_size, embedding_size]
        _, (ht, ct) = self.encoder(enc_input)
        # en_ht: [num_layers * num_directions, batch_size, hidden_size]
        de_output = self.decoder((ht, ct), enc_input)
        # de_output: [seq_len, batch_size, in_features]
        output = self.fc(de_output)
        # output: [seq_len, batch_size, hidden_size]
        output = output.permute(1, 0, 2)
        return output


# 从梅尔频谱图集合构建数据加载器
batch_size = 32
train_dataset = TensorDataset(train_source, train_target)
test_dataset = TensorDataset(test_source, test_target)
train_loader = DataLoader(train_dataset, batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size, shuffle=True)

train_loss_table = []
test_loss_table = []

# 将架构实例化
net = Net().cuda()

# 损失函数
loss_fn = nn.MSELoss()
```

```
# 定义优化器结构
optimizer = torch.optim.Adam(net.parameters(), lr=1e-5)

# 开始训练流程，默认对所有训练集数据迭代 10 轮
for epoch in range(10):
    train_loss = 0                          # 训练集误差
    # 开始对所有数据迭代
    net.train()                             # 将模型调整为训练模式
    for source, target in train_loader:
        # 将数据加载进显存，以便使用 GPU 加速运算
        source = source.cuda()
        target = target.cuda()
        y_pred = net(source)
        # print(y_pred.shape, target.shape)
        y_pred = y_pred.reshape(y_pred.shape[0], -1)
        target = target.reshape(target.shape[0], -1)
        loss = loss_fn(y_pred, target)      # 计算合成音频与目标音频的差异
        ↪ 度
        train_loss += loss.item()           # 将本轮的误差累加
        loss.backward()                     # 反向传播误差值，计算偏导数
        optimizer.step()                    # 利用偏导数更新神经网络参数
    train_loss_table.append(train_loss)

    # 测试流程
    test_loss = 0                   # 测试集误差
    net.eval()                      # 将模型调整为测试模式
    with torch.no_grad():           # 关闭偏导数计算
        for source, target in test_loader:
            source = source.cuda()
            target = target.cuda()
            y_pred = net(source)
            y_pred = y_pred.reshape(y_pred.shape[0], -1)
            target = target.reshape(target.shape[0], -1)
            loss = loss_fn(y_pred, target)
            test_loss += loss.item()
    test_loss_table.append(test_loss)
    print(f'[Epoch {epoch}]: Train Loss: {train_loss / n_train:.10f},
    ↪ Test Loss: {test_loss / n_test:.10f}')
```

```
# 绘制损失函数图像
plt.figure(figsize=(10, 6))
plt.plot(train_loss_table, color='blue', label='Train Loss')
plt.plot(test_loss_table, color='orange', label='Test Loss')
plt.title('Training and Testing Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.savefig('loss.jpg')

# 将测试集输出保存到文件
scipy.io.savemat('output.mat', {'output':
↪  y_pred.cpu().detach().numpy()})
```

D 音频重建模块（Python）

```
import librosa
import scipy.io
y_pred = scipy.io.loadmat('output.mat')['output']
mels = []
for i in range(y_pred.shape[0]):
    mel = y_pred[i].cpu().detach().numpy().reshape(1024, 32)
    mels.append(mel)
mels = np.array(mels)
audio = librosa.feature.inverse.mel_to_audio(mels, sr=16000)

for i in range(audio.shape[0]):
    sf.write(f'{i}.mp3', audio[i], 16000)
```