实验三：基于识别合成框架的语音转换

1 任务一：提取 BNF 与声学参数

1.1 简要说明 BNFs 是什么，其在基于识别合成的语音转换框架中起到什么作用?

(1)

BNFs（BottleNeck Features，瓶颈特征）通常指的是在神经网络架构中，尤其是语音识别和语音合成的任务中，使用瓶颈层提取的特征。瓶颈层是一种通过将输入特征压缩到低维度空间，从而提取出最具判别性、最重要的特征的机制。通过这一过程，网络能够聚焦于输入数据的关键信息，去除冗余和噪声。

(2)

基于识别合成框架的主要思路是先通过语音识别将语音转化为某种中间表示（如特征或隐层状态），然后再利用这些中间表示进行语音合成。在这一框架中，BNFs的作用如下：

- **压缩与提取特征**：BNF通过神经网络的瓶颈层进行特征压缩，提取关键信息，并去除冗余数据。这个低维度的表示能够保留原始语音中重要的语音特征，例如发音、语调、音色等，而剔除不必要的噪声和冗余信息。
- **有效传递中间表示**：在识别合成框架中，语音识别阶段产生的中间表示（如BNFs）被传递到语音合成阶段。这些特征能够有效地传递识别到的语音信息，以便合成系统生成更自然的语音。
- **跨任务共享与迁移**：通过在识别和合成过程中都使用BNFs，能够实现识别和合成任务之间的知识共享和迁移。例如，在多任务学习中，BNFs不仅帮助识别和合成任务协同工作，还可以增强语音模型的泛化能力。
- **抑制噪声和冗余信息**：BNFs特征通常有助于提高系统在复杂环境下的鲁棒性。通过在神经网络中学习到的瓶颈层特征，系统能更好地从噪声数据中提取有效信息，增强合成语音的清晰度和自然度。

1.2 理解声学参数提取的过程，为 `hparam.py` 中 class Audio 的主要参数添加注释，说明该参数的意义，可在实验报告中截图或者表格展示。

```
class Audio:
    num_mels = 80             # Mel谱频率带数量/特征参数维度
    ppg_dim = 351             # PPG（语音后验概率图）特征维度
    bn_dim = 256              # 瓶颈层特征维度
    num_freq = 1025           # 线性谱频率带数量/特征参数维度
    min_mel_freq = 30.        # 最低Mel频率
    max_mel_freq = 7600.      # 最高Mel频率
    sample_rate = 16000       # 采样率
    frame_length_ms = 25      # 每帧音频的时间长度
    frame_shift_ms = 10       # 连续帧之间的间隔
    upper_f0 = 500.           # 基本频率（F0）的上限
    lower_f0 = 30.            # 基本频率（F0）的下限
    n_mfcc = 13               # 提取的 MFCC（梅尔频率倒谱系数）特征的数量
    preemphasize = 0.97       # 预加重系数：用于强调音频信号中的高频成分
    min_level_db = -80.0      # 频谱图的最小值
    ref_level_db = 20.0       # 频谱图归一化的参考值
    max_abs_value = 1.        # 归一化频谱图允许的最大绝对值
    symmetric_specs = False # 是否使用围绕0对称的频谱图值
    griffin_lim_iters = 60  # Griffin-Lim 算法的迭代次数（用于波形重建）
    power = 1.5               # 频谱图的幂次方
    center = True             # 是否对音频帧进行居中处理
```

1.3 Mel 谱和线性谱的提取过程有什么差异？它们之间是什么关系？

(1)

线性谱是直接计算信号的傅里叶变换，然后计算得到的频谱。这个频谱是基于频率轴的线性尺度。线性谱的提取过程通常包括以下步骤：

- **分帧**：将音频信号分成短时帧，每一帧表示信号在该时间窗口内的特征。
- **加窗**：对每一帧应用窗函数，以减少频谱泄漏。
- **傅里叶变换**：对每一帧信号进行快速傅里叶变换，得到每一帧的频谱。
- **形成幅度谱**：取傅里叶变换的幅度部分，得到该帧的频谱表示。

Mel谱的提取过程类似于线性谱，但它额外在频率轴上应用了Mel尺度的变换。Mel尺度是一种接近人耳感知的频率尺度，在低频区域频率分辨率较高，在高频区域频率分辨率较低。Mel谱的提取过程通常包括：

- **分帧、加窗、傅里叶变换**：与线性谱相同。

- **Mel滤波器组**：将频谱通过一个Mel滤波器组进行处理，将线性频谱转换为Mel频谱。滤波器组的设计使得低频部分的频率分辨率更高，而高频部分的分辨率较低。
- **对数转换**：有时对Mel谱应用对数变换，模拟人耳对声音的感知。
- **形成Mel谱图**：得到的对数能量值构成了Mel谱图。

(2)
Mel谱和线性谱的关系：

- **频率轴的尺度不同**：线性谱在频率轴上均匀分布，而Mel谱在低频部分具有更高的分辨率，在高频部分分辨率较低。
- **Mel谱的频率分辨率低于线性谱**：由于Mel滤波器组将频率压缩到Mel尺度上，Mel谱的频率分辨率较低，尤其是在高频区域。
- **Mel 谱是对线性谱的进一步处理**：Mel谱是在线性谱的基础上，通过应用Mel滤波器组和取对数能量来得到的。
- **不同的应用重点**：线性谱通常在需要精确频率信息的场合使用（如音乐分析），而 Mel 谱更多用于语音识别等应用，因为它更贴近人类对声音的感知方式。

1.4 提取出来的线性谱和Mel谱各是多少维的特征参数？Mel谱的频率范围是多少？

根据 `hparam.py` 中的参数：
(1)
线性谱是1025维的特征参数；Mel谱是80维的特征参数。
(2)
Mel谱的频率范围是从30Hz到7600Hz。

1.5 在语音转换中BNFs提供了内容信息，基频参数F0提供了什么信息？为什么要考虑基频参数F0？

(1)
基频参数F0提供了语音信号的基本音高信息。
(2)
在进行语音转换时，考虑基频F0很重要，因为它不仅有助于保持语音的自然性和连贯性，还有助于保留或转换说话者的个性特征和情感表达。通过调整基频，可以更好地模仿目标说话者的声音特征，使转换后的语音听起来更加真实和自然。

1.6 实验中将数据集划分为了训练集、验证集、测试集。它们之间的默认划分比例是多少？

根据 `hparam.py` 中的参数，训练集、验证集和测试集之间的默认划分比例是$85:10:5$，即$17:2:1$。

2 任务二：训练并测试特定目标说话人的语音转换模型

2.1 DataLoader的调用参数batch_size、shuffle、num_workers、collate_fn分别是什么含义?

- batch_size：决定了每个数据批次的大小。在每一次迭代中，DataLoader会提取一个包含batch_size个样本的批次。
- shuffle：是否在每个epoch开始时打乱数据集。
- num_workers：指定加载数据时使用的子进程数，使用多个子进程并行加载数据可以加快数据加载速度。
- collate_fn：指定如何将多个数据样本合并成一个批次。

2.2 阅读 `train_to_one.py`，找出转换模型的定义，并根据 `models/models.py` 说明转换模型的网络结构，给出网络的基本结构图。

(1)

`train_to_one.py` 中转换模型的定义如以下代码所示：

```
# set up model
model = BLSTMConversionModel(in_channels=hps.Audio.bn_dim + 2,
                             out_channels=hps.Audio.num_mels,
                             lstm_hidden=hps.BLSTMConversionModel.lstm_hidden)
```

(2)

`models/models.py` 中定义的转换模型网络结构如以下代码所示：

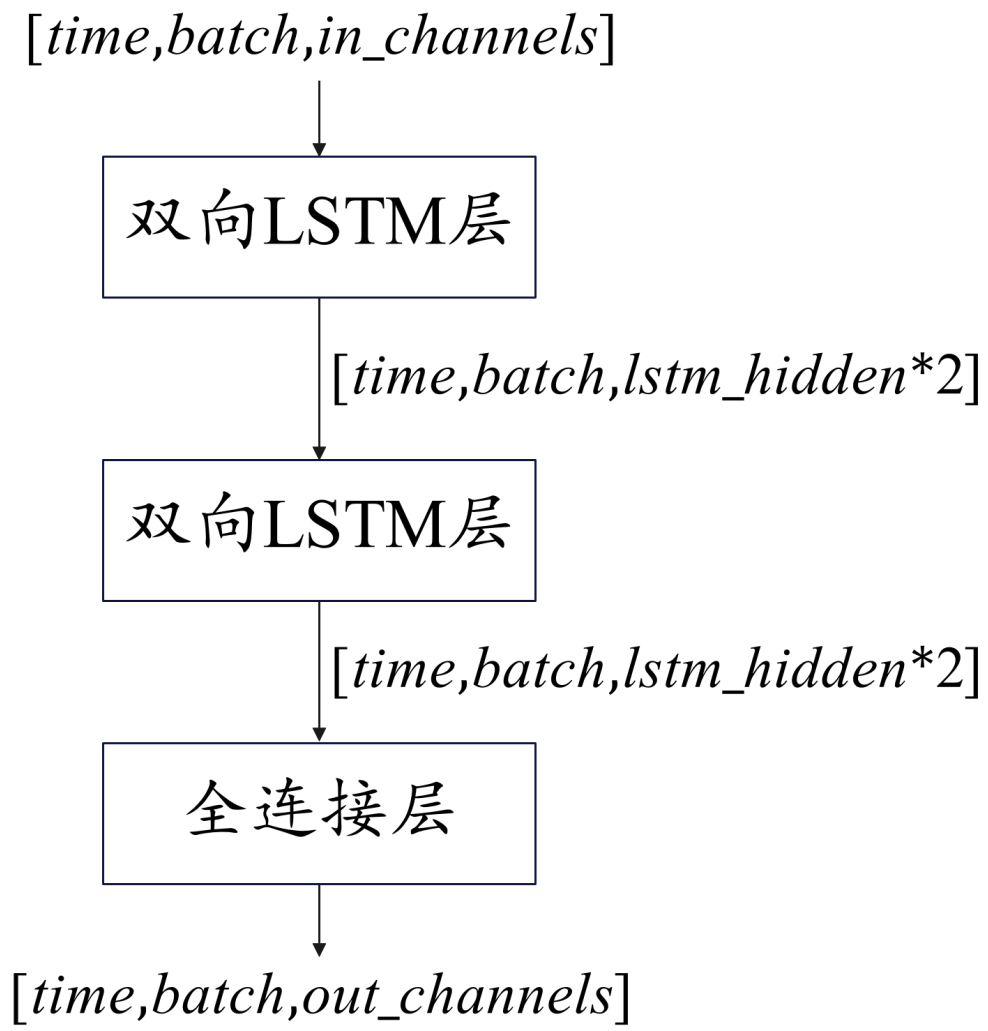
```
class BLSTMConversionModel(nn.Module):
    def __init__(self, in_channels, out_channels, lstm_hidden):
        super(BLSTMConversionModel, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.lstm_hidden = lstm_hidden
        self.blstm1 = nn.LSTM(input_size=in_channels,
                              hidden_size=lstm_hidden,
                              bidirectional=True)
        self.blstm2 = nn.LSTM(input_size=lstm_hidden * 2,
                              hidden_size=lstm_hidden,
                              bidirectional=True)
        self.out_projection = nn.Linear(in_features=2 * lstm_hidden,
                                        out_features=out_channels)

    def forward(self, x):
        # pass to the 1st BLSTM layer
        blstm1_out, _ = self.blstm1(x)
        # pass to the 2nd BLSTM layer
        blstm2_out, _ = self.blstm2(blstm1_out)
        # project to the output dimension
        outputs = self.out_projection(blstm2_out)
        return outputs
```

故转换模型的网络结构从输入到输出为：

- **双向LSTM层1**：用于学习序列的时序依赖性。
- **双向LSTM层2**：用于学习序列的时序依赖性。
- **全连接层**：将双向LSTM输出的特征映射到最终的目标特征空间。

网络的基本结构图为



2.3 进而说明转换模型的输入、输出参数分别是什么？输入、输出参数的维数是多少？

(1)

输入参数是拼接后的特征，包含以下两个主要部分（会在模型的输入层被拼接起来）：

- **瓶颈特征（BNF）**：通常从一个预先训练的神经网络中提取的语音内容特征。
- **基本频率（F0）**：表示语音的基频，反映了音高信息。

输出参数是经过转换后的Mel频谱特征，这些特征表示了转换后的音频的频域信息。

(2)

输入参数的维度是[time,batch,in_channels]，其中in_channels具体值为 `hps.Audio.bn_dim + 2 = 258`；
输出参数的维度是[time,batch,out_channels]，其中out_channels具体值为 `hps.Audio.num_mels = 80` 。

2.4 说明转换模型的训练过程，如何进行前向计算？使用了什么损失函数？损失函数是怎么计算的？如何进行误差反向传播？

(1)

前向计算是指输入通过模型进行处理，并生成输出的过程。具体来说，以下是前向计算的步骤：

- **拼接特征构建输入：** 输入特征 `inputs` 是通过将瓶颈特征（`bnf`）和基本频率（`f0`）拼接得到的。
- **通过第一个 BLSTM 层：** 输入张量 `x` 首先通过一个双向 LSTM（`blstm1`）进行处理。该层会根据输入特征的维度（`in_channels`）生成一个隐层输出（`blstm1_out`）。由于是双向 LSTM，因此输出的维度是 `lstm_hidden * 2`，即隐层维度 `lstm_hidden` 的两倍。
- **通过第二个 BLSTM 层：** 然后，`blstm1_out` 被送入第二个双向 LSTM 层（`blstm2`）。这会进一步处理该输出，并得到一个新的隐藏状态（`blstm2_out`）。与第一层一样，输出的维度为 `lstm_hidden * 2`。
- **通过全连接层：** 最后，`blstm2_out` 经过全连接层（`out_projection`），将输出特征映射到目标维度 `out_channels`。
- **输出：** 通过前向计算，模型最终输出的是转换后的 Mel 频谱（`outputs`）。

(2)

模型使用自定义的 `masked_mse_loss` 函数作为损失函数。

(3)

损失函数的计算代码如下：

```
def masked_mse_loss(inputs: torch.Tensor, targets: torch.Tensor, lengths: torch.Tensor):
    if lengths is None:
        return nn.MSELoss()(inputs, targets)
    else:
        max_len = max(lengths.cpu().numpy())
        mask = torch.arange(max_len).expand([len(lengths), max_len]).to(device) < lengths.unsque
        mask = mask.to(dtype=torch.float32)
        mse_loss = torch.mean(
            torch.sum(torch.mean((inputs - targets) ** 2, dim=2) * mask,
                      dim=1) / lengths.to(dtype=torch.float32))
        return mse_loss
```

`masked_mse_loss` 是一个自定义的均方误差损失函数，它用于计算两个张量之间的均方误差，同时考虑到填充部分的掩蔽。其目的是避免对填充部分计算损失，只对有效的时间步（即非填充部分）进行损失计算。具体计算步骤如下：

- 如果没有填充信息，则计算标准的均方误差损失。

- 如果有填充信息，则根据其提供的有效时间步长度，构造掩蔽，确保只有有效的时间步参与损失计算，忽略填充部分。
- 最后，对每个样本进行归一化，并对所有样本的损失进行平均。

(4)

误差反向传播的目的是通过计算损失函数相对于模型参数的梯度，然后更新模型的参数，使得损失最小化。具体步骤如下：

- **清除先前的梯度：** 在每次前向传播之前，先使用 `optimizer.zero_grad()` 清除优化器中的梯度信息。
- **计算损失并进行反向传播：** 通过 `loss.backward()` 来计算损失函数相对于模型参数的梯度。同时， `loss.backward()` 会执行反向传播，计算出损失函数对于模型中每个参数（如 LSTM 的权重、偏置等）的梯度。
- **更新模型参数：** 使用优化器来更新模型的参数。通过 `optimizer.step()` 执行一步优化，利用计算出的梯度调整模型参数。

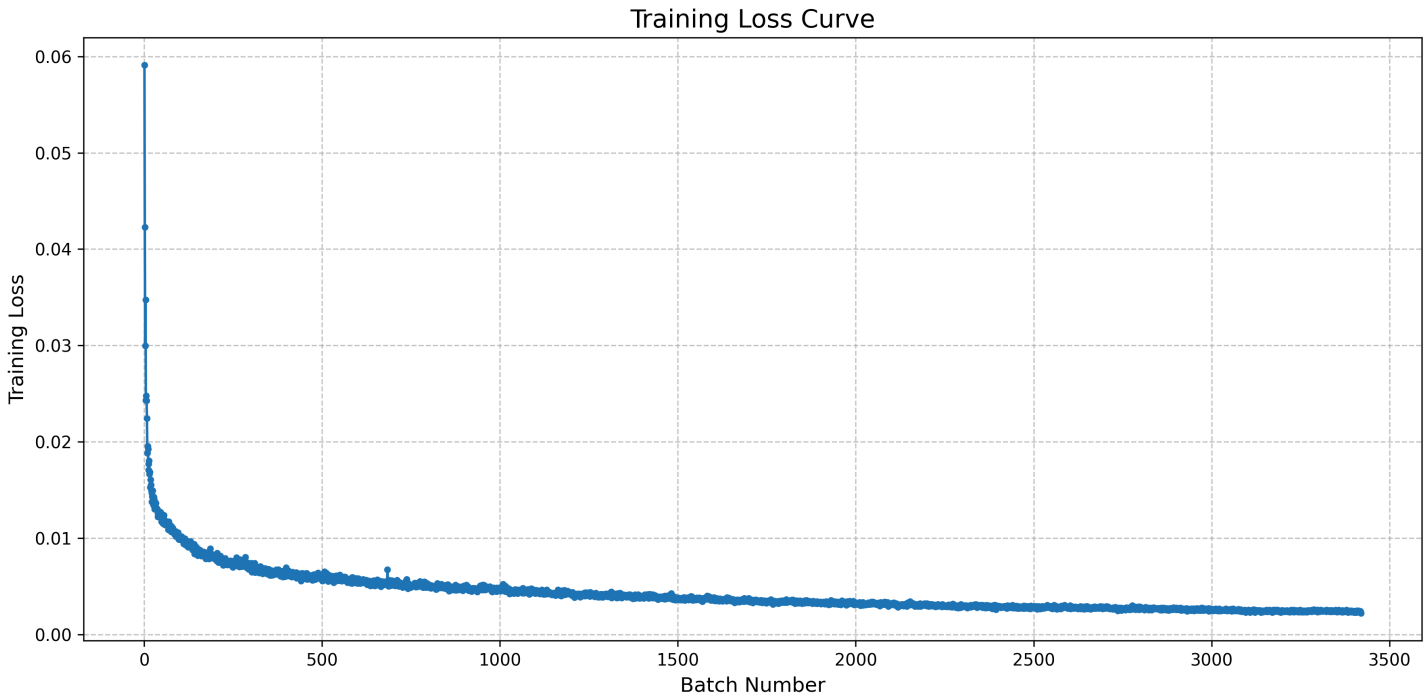
2.5 给出模型训练时的损失函数曲线；模型训练了多少个 epoch？训练完成后，最终（对应最后一个 epoch）的平均训练 MSE loss 是多少?

使用的训练指令为

```
nohup python train_to_one.py \
--model_dir pretrained_model/asr_model/bzn_res/ \
--test_dir result/bzn_res/ \
--data_dir save_data/bzn/ > run_3.log 2>&1 &
```

(1)

编写 `draw_loss.py` ， 根据训练日志 `run_2.log` 绘制模型训练时的损失函数曲线如下所示：



(2)
模型训练了60个epoch。
(3)
最终的平均训练 MSE loss 是0.00238。

2.6 模型训练的结果保存在什么地方?

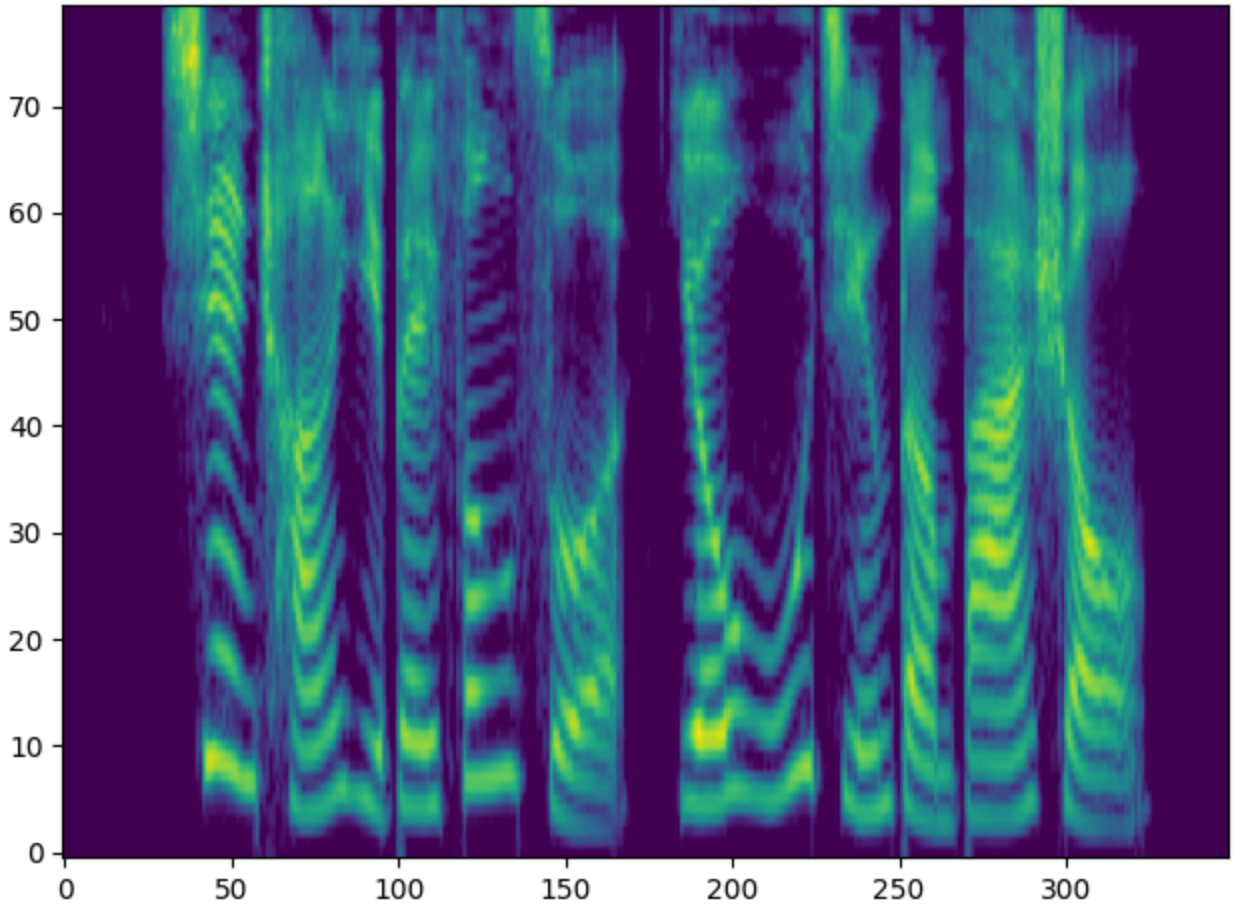
本实验模型训练的结果保存在 `pretrained_model/asr_model/bzn/` 中，通过训练指令中的 `--model_dir` 参数设置。

2.7 基于上述训练好的模型，在验证集上进行验证。在验证集上的 MSE loss 是多少?

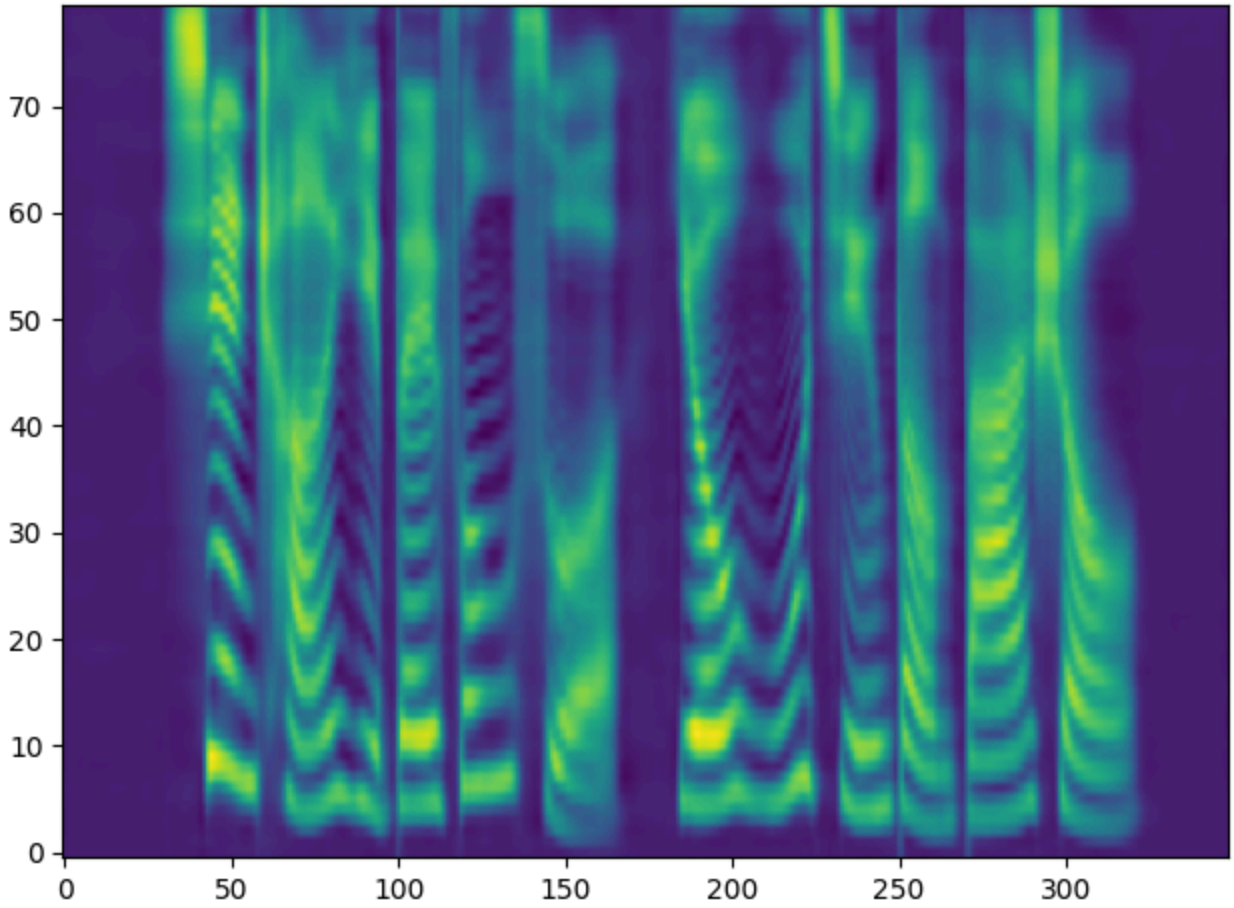
根据训练日志 `run_2.log` ，在验证集上的MSE loss是0.00613。

2.8 基于上述训练好的模型，在测试集上进行测试。针对某句测试用例，得到该用例的真实语音以及预测的转换语音；给出该测试用例的真实 Mel 谱、预测 Mel 谱的图，简单分析它们之间的区别，并计算这两个 Mel 谱之间的均方差 MSE 距离。

测试用例选择 `bzn/001770.wav` ，得到该用例的真实语音以及预测的转换语音。
该测试用例的真实Mel谱如下图所示:



该测试用例的预测Mel谱如下图所示：



由两图对比可以看出，真实 Mel 谱和预测 Mel 谱大致分布相似，但预测 Mel 谱在全频段，尤其是高频段上，显示出的谐波不那么清晰，可能说明转换时并未很好地获取更精细的语音特征。
这两个 Mel 谱之间的均方差 MSE 距离为0.0050。

2.9 分析在 `inference_to_one.py` 中所调用的 inv_mel_spectrogram 函数，说明将Mel谱恢复为语音波形（Speech Waveform）的基本过程，理解声码器（Vocoder）的作用。

该函数的定义如下:

```
def inv_mel_spectrogram(mel_spectrogram):
    S = _mel_to_linear(_db_to_amp( #1e-10 25
        _denormalize(mel_spectrogram)+audio_hp.ref_level_db))
    return _griffin_lim(S ** audio_hp.power)
```

该函数的目的是将 Mel 频谱图 `mel_spectrogram` 转换回语音波形。整体流程为：

- **反归一化**：通过 `_denormalize` 将 Mel 频谱图恢复到原始的幅度范围。
- **对数幅度到线性幅度转换**：通过 `_db_to_amp` 将幅度从对数（dB）尺度转换回线性尺度。
- **Mel 频谱到线性频谱转换**：通过 `_mel_to_linear` 将 Mel 频谱图反向变换为线性频谱图。

- **利用 Griffin-Lim 算法重建波形**：通过 `_griffin_lim` 将频谱图转换回时域波形。

各个子函数的基本功能为：

- `_denormalize(mel_spectrogram)`：该函数的作用是对 Mel 频谱图进行反归一化处理。在训练过程中，通常会对输入的 Mel 频谱图进行归一化，以便于神经网络学习。因此，在将 Mel 频谱图作为输入时，需要对其进行反归一化，以恢复其原始的幅度范围。
- `_db_to_amp()`：该函数的作用是将以分贝表示的幅度转换为线性幅度。Mel 频谱图中的幅度通常以对数形式表示，而实际的信号幅度是线性的，因此需要将其转换回线性幅度。
- `_mel_to_linear()`：该函数的作用是将 Mel 频谱图转换为线性频谱图。Mel 频谱图是通过 Mel 频率尺度（非线性尺度）对原始频谱进行压缩得到的，因此需要反向操作来将 Mel 频谱图转换为传统的线性频谱图。
- `_griffin_lim()`：这是一个用于信号重建的算法，通常用于从幅度谱和估算的相位谱重建时域信号。Griffin-Lim 算法的核心思想是：给定幅度谱，通过反复调整相位来逐渐逼近原始波形。

声码器的核心作用是在这一步完成，将 Mel 频谱图转换回可听的语音波形。

2.10 结合 inference_to_one.py，说明转换阶段由输入源说话人的语音到输出特定目标说话人语音的整体流程。

转换阶段的整体流程如下：

- **输入源语音**：加载并预处理源语音（包括预加重）。
- **提取源语音特征**：提取源语音的高级特征（BNFs）、基频（F0）和 Mel 频谱图。
- **构造输入**：将 BNFs 和 F0 特征结合成输入，作为语音转换模型的输入。
- **语音转换模型推断**：使用预训练的 BLSTM 模型进行推断，生成目标说话人的 Mel 频谱图。
- **波形合成**：将预测的 Mel 频谱图恢复为时域波形，得到目标语音。

2.11 选取上述训练好的模型所对应的 checkpoint 文件（ckpt 参数），给提供数据中某个特定说话人的某句语音作为输入（src_wav 参数），进行语音转换得到转换后的语音（save_dir 参数）。

使用的推理命令为：

```
python inference_to_one.py \
--src_wav dataset/bzn/001770.wav \
--ckpt pretrained_model/asr_model/bzn/bnf-vc-to-one-59.pt \
--save_dir result/bzn_inference/
```

2.12 自己录制一段语音，并作为模型的输入（src_wav 参数），重复上述（11）的语音转换过程，得到转换后的语音。

录制了 `record.wav` 。
使用的推理命令为：

```
python inference_to_one.py \
--src_wav record.wav \
--ckpt pretrained_model/asr_model/bzn/bnf-vc-to-one-59.pt \
--save_dir result/bzn_inference/
```

2.13 在实验报告中对（11）（12）中的转换后的语音的效果进行简单分析。

对于2.11中转换后的语音 `bzn/001770.wav` ： 转换前是 mst-female 的音色，转换后与 bzn 说话的声音很相似，能清晰辨别文本内容，基本没有杂音。
对于2.12中转换后的语音 `record.wav` ： 转换效果和（11）类似，整体转换效果较好，能清晰地听出文本内容和录制的音频内容相同，且杂音和噪声较小。

2.14 简要说明多对一语音转换（any-to-one VC）为什么能够实现给定任意源说话人的“多”对一的语音转换，关键是什么?

多对一语音转换（any-to-one VC）是指从多个不同的源说话人的语音转换为同一个目标说话人的语音，关键在于提取说话人无关的特征（如内容和音高）以及学习将这些特征映射到目标说话人特征的映射函数。详细分析如下：

（1）说话人无关的特征提取：

- 通过使用说话人无关的特征，例如从源语音中提取的高层语音特征（如 BNFs）或音高（F0），模型能够聚焦于语音的内容（如说话的内容、韵律等），而不是语音的个体差异（如音色、口音等）。
- 这些特征能够捕捉到语音中的语言内容和音高等特征，而不受说话人差异的影响，保证了不同源说话人的语音能够转化为相同的目标说话人的语音。

（2）目标说话人的特征建模：

- 在多对一转换中，目标说话人的特征（如音色、音调、韵律等）被建模成一个统一的目标风格。转换模型需要学习如何将不同源说话人的语音特征映射到这个目标风格。
- 通过训练，模型能够掌握从源说话人特征到目标说话人风格的映射，从而在转换过程中保持一致的目标说话人的音色和语音特征。

（3）模型的泛化能力：

- 多对一转换模型必须具备较强的泛化能力，即它不仅能处理一个特定的源说话人，而是能够接受任何源说话人的输入，并将其转换为目标说话人的语音。
- 为此，模型需要对源说话人的个体差异具有一定的鲁棒性，能够提取与说话人无关的高层次语音特征。

3 任务三：探究残差网络结构对转换性能的影响

3.1 根据残差网络结构的有关原理，确定 ResidualNet 的某种特定结构（考虑到数据量的大小，建议不要使用很复杂的网络结构），实现 `models/models.py` 中 ResidualNet 类的具体代码；并对 `models/models.py` 中 BLSTMResConversionModel 类的相应部分进行修改。

ResidualNet 类的代码实现如下：

```
class ResidualNet(nn.Module):
    def __init__(self, channels, num_blocks):
        super(ResidualNet, self).__init__()
        self.channels = channels
        # define your module components below
        # e.g. self.dense_layer = nn.Linear(in_features=channels, out_features=channels)
        self.residual_blocks = nn.ModuleList([
            self._make_residual_block(channels) for _ in range(num_blocks)
        ])

        # Final layer to adjust residual contribution
        self.final_layer = nn.Linear(channels, channels)

    def _make_residual_block(self, channels):
        return nn.Sequential(
            nn.LayerNorm(channels),
            nn.Linear(channels, channels),
            nn.ReLU(),
            nn.Linear(channels, channels)
        )

    def forward(self, x):
        # define your inference process below
        residual = x
        for block in self.residual_blocks:
            residual = block(residual) + residual
        residual = self.final_layer(residual)
        return residual
```

BLSTMResConversionModel 类的代码实现如下：

```
class BLSTMResConversionModel(nn.Module):
    def __init__(self, in_channels, out_channels, lstm_hidden, num_res_blocks=3):
        super(BLSTMResConversionModel, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.lstm_hidden = lstm_hidden
        self.blstm1 = nn.LSTM(input_size=in_channels,
                              hidden_size=lstm_hidden,
                              bidirectional=True)
        self.blstm2 = nn.LSTM(input_size=lstm_hidden * 2,
                              hidden_size=lstm_hidden,
                              bidirectional=True)
        self.out_projection = nn.Linear(in_features=2 * lstm_hidden,
                                        out_features=out_channels)
        self.resnet = ResidualNet(out_channels, num_blocks=num_res_blocks)

    def forward(self, x):
        # pass to the 1st BLSTM layer
        blstm1_out, _ = self.blstm1(x)
        # pass to the 2nd BLSTM layer
        blstm2_out, _ = self.blstm2(blstm1_out)
        # project to the output dimension
        initial_outs = self.out_projection(blstm2_out)
        residual = self.resnet(initial_outs)
        final_outs = initial_outs + residual # define the final outputs here
        return final_outs
```

3.2 说明所实现的 ResidualNet 的具体结构，并给出网络的基本结构图。

(1)

实现的ResidualNet按先后顺序由串联的3个残差块和1个最终层组成，具体细节如下：

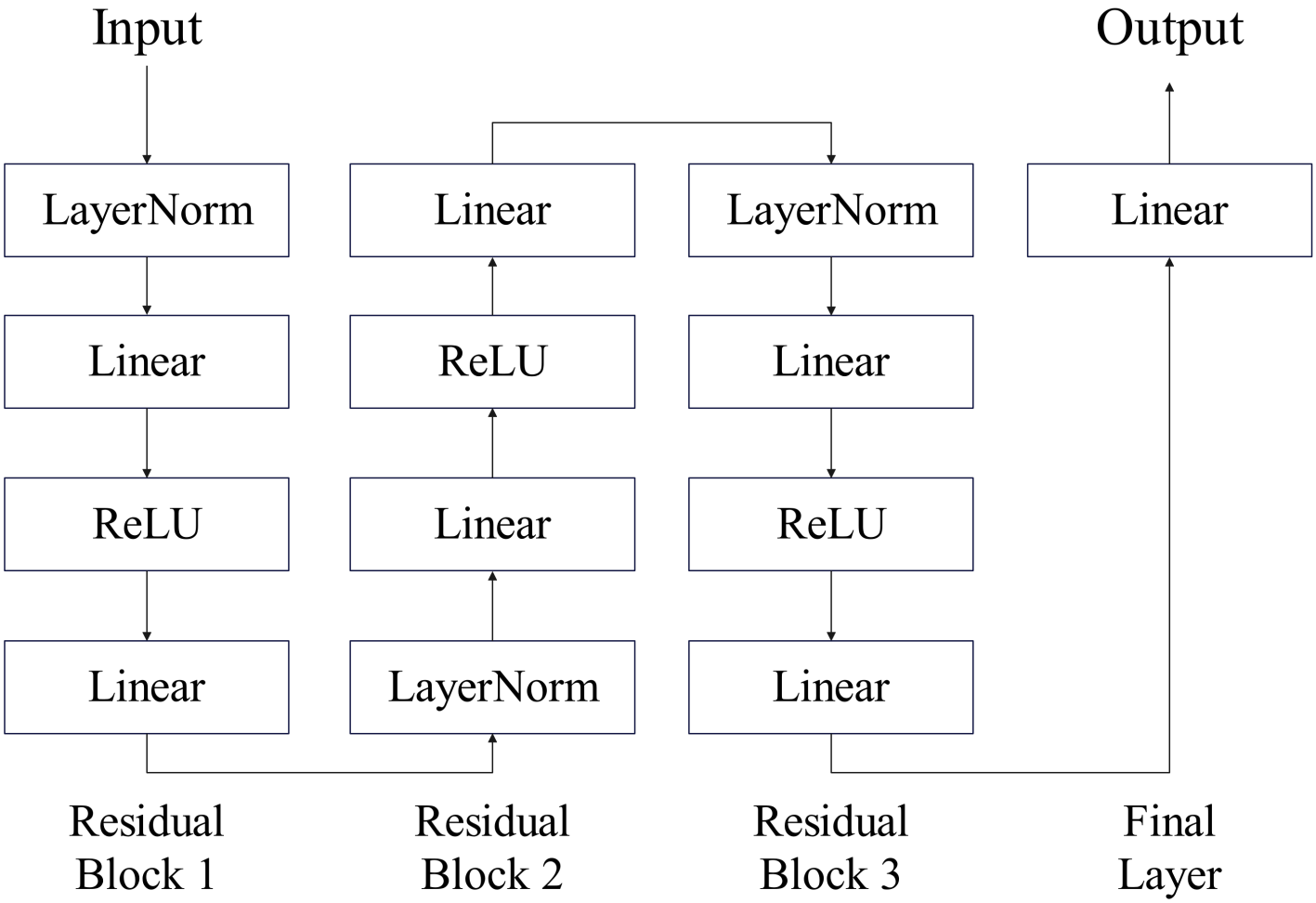
- **残差块：**
  - 该架构的主要特征是使用残差块。这些块允许模型学习残差（块输入和输出之间的差异），而不是学习整个映射。
  - 每个残差块由两个带有ReLU激活函数的线性层 组成，并且在每个块的输入中应用了层归一化。

  每个块中的过程如下：
  - 输入首先进行层归一化，然后进入一个线性层（将输入映射到相同的通道维度）。
  - 结果通过ReLU激活函数。

- 接着是另一个线性层，同样输出相同的通道维度。
- 输出与原始输入相加（即残差连接）。

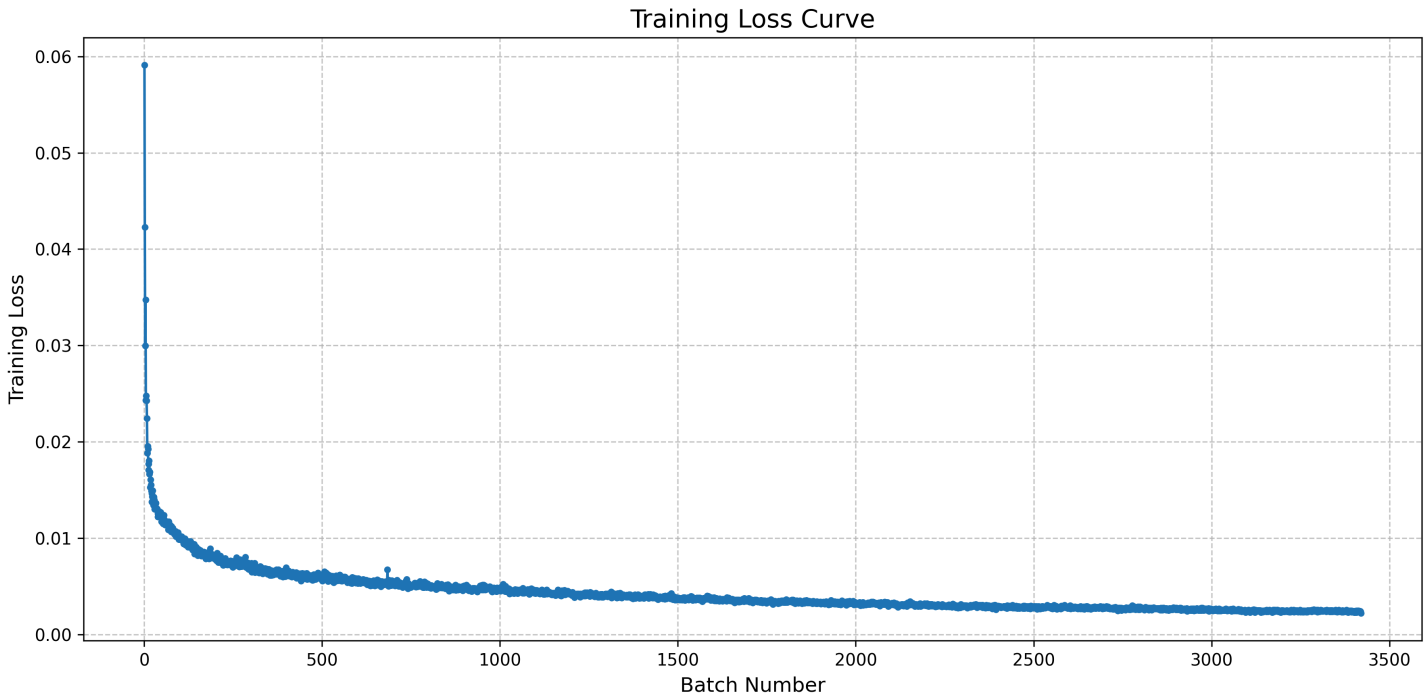
- **最终层：**
  - 经过所有残差块之后，网络应用一个最终的线性层来调整残差学习过程的输出，以确保输出符合Mel 频谱图生成任务的要求。
- **前向传播：**
  - 在 `forward` 方法中，首先将输入 `x` 赋值给残差，然后将其传递给每个残差块。每个残差块处理完后，将处理结果与其输入进行拼接。
  - 经过所有块的处理后，最后的残差信息经过最终层。

(2)
网络的基本结构图如下：

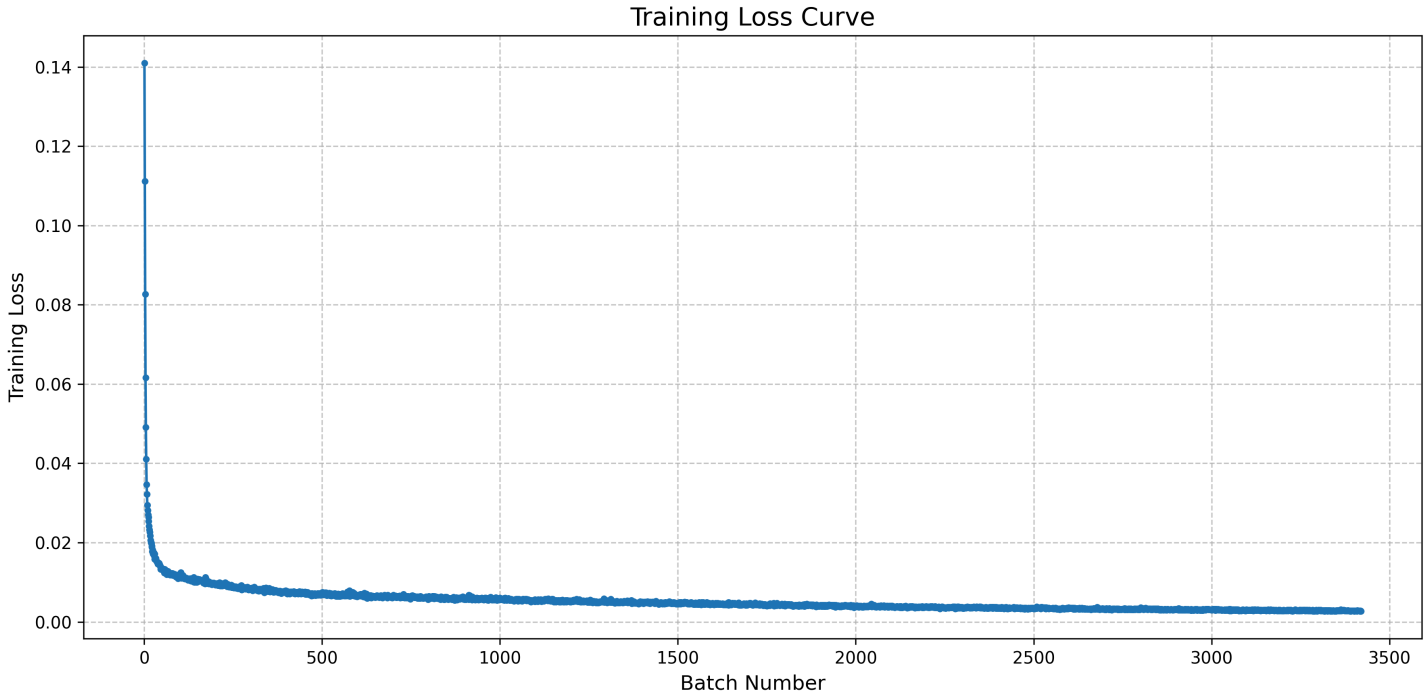


3.3 进行消融实验，探究上述任务二中无残差网络结构的转换模型、与本任务三中实现的有残差网络结构的转换模型的性能差别。

(1)**模型在训练过程中的MSE loss曲线的差异**
任务二模型的MSE loss曲线如下：(最后一轮平均训练 MSE loss 是0.00238)



任务三模型的MSE loss曲线如下：(最后一轮平均训练 MSE loss 是0.00282)



可以发现，在训练过程中，任务二模型的训练集loss降得比任务三模型快一些。

(2)模型训练完成后在验证集上的MSE loss值的差异

任务二模型训练完成后在验证集上的MSE loss值为0.00613;

任务三模型训练完成后在验证集上的MSE loss值为0.00596;

可以发现，任务二模型在验证集上的loss值比任务三模型更高，说明任务二模型较任务三模型更易陷入过拟合。

(3)同样测试用例下，转换语音的差异

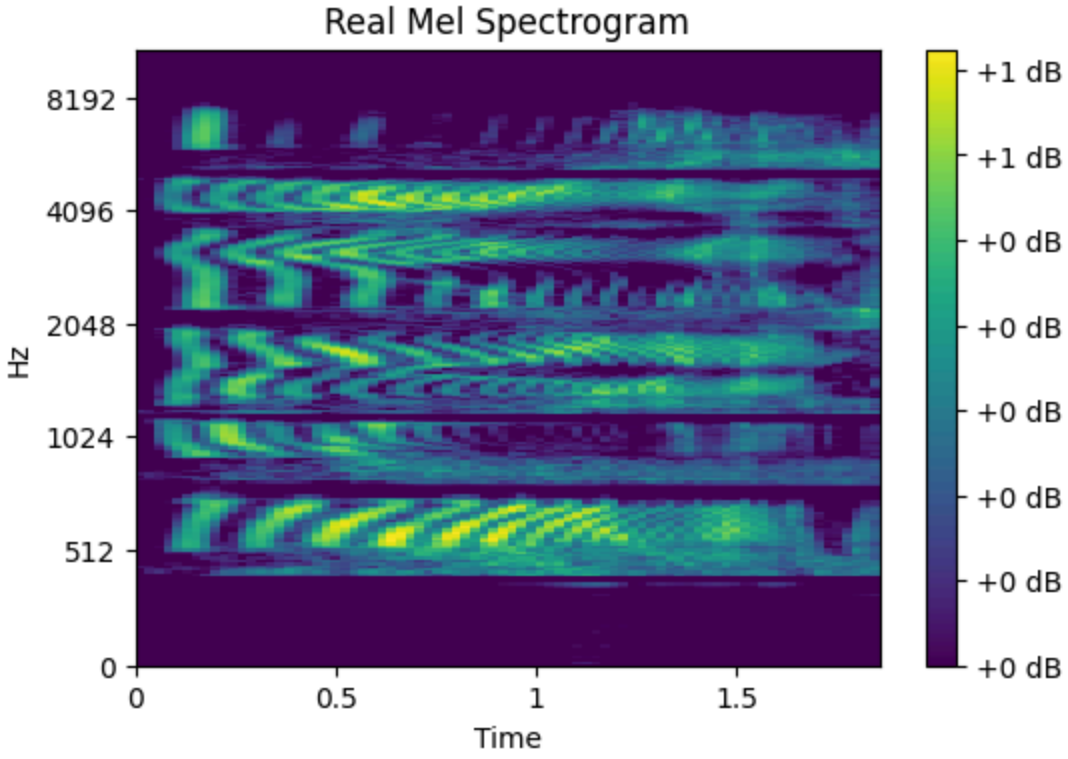
以 `bzn/000001.wav` 为测试用例。

效果都挺逼真的，听不出显著的差异（不过可能是因为测试用例对模型有点简单了）。

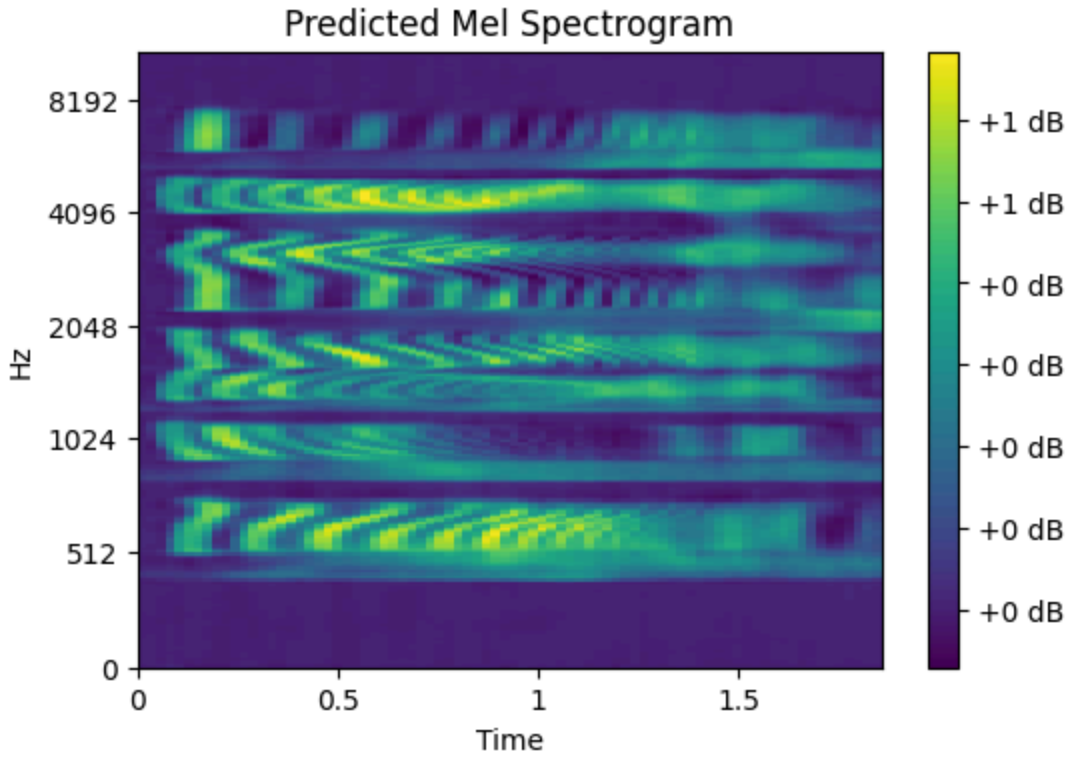
(4)同样测试用例下，预测的Mel谱及谱图的差异

以 `bzn/000001.wav` 为测试用例。

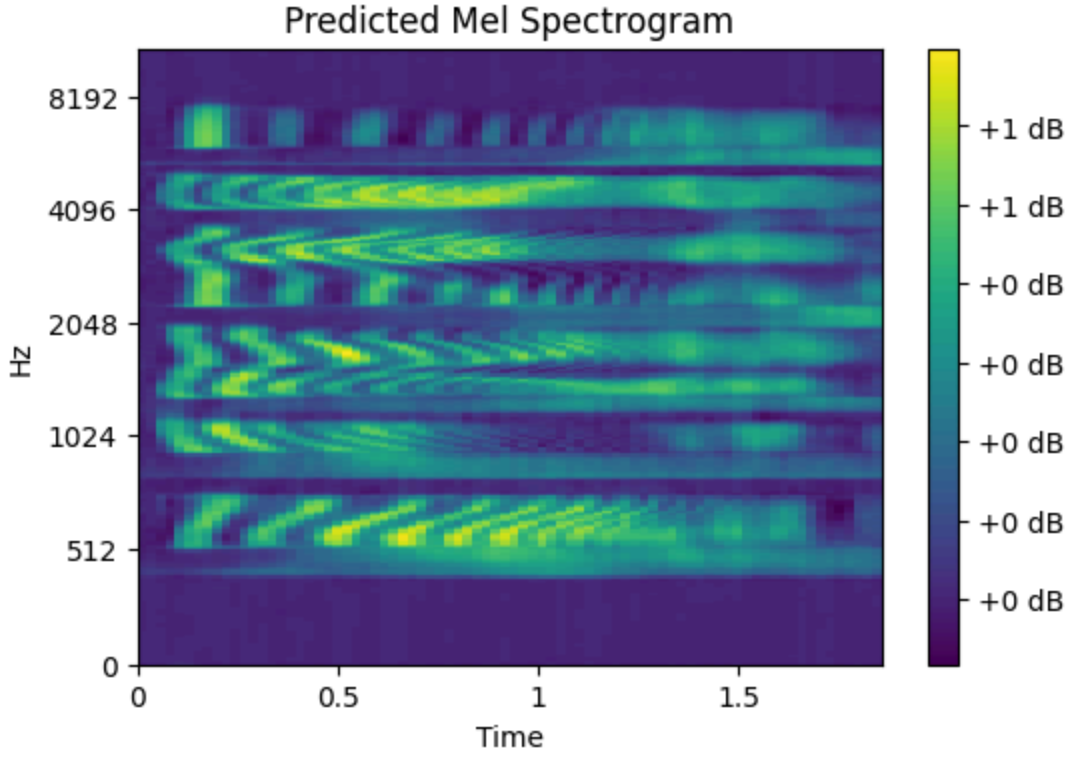
该测试用例的真实Mel谱如下图所示：



任务二模型预测的Mel谱如下图所示：（与真实Mel谱之间的均方差 MSE 距离为0.00533）



任务三模型预测的Mel谱如下图所示：（与真实Mel谱之间的均方差 MSE 距离为0.00530）



对比两个预测Mel谱可知：任务三模型得到的预测 Mel 谱质量比任务二模型得到的更好，与真实Mel谱差异更小，谐波也更清晰。

综上，有残差网络结构的转换模型性能优于无残差网络结构的转换模型。但是，目前测试的数据量太少，不能排除偶然性，该结论可能有待商榷。

4 任务四：增加说话人嵌入网络，实现多目标说话人的语音转换

4.1 根据 GitHub repository 的说明文档中的要求（Any-to-Many → Feature Extraction），运行相关命令进行特征提取。

运行过程截图如下：

```
(audio) root@LAPTOP-05QND3DR:/mnt/c/Users/ysn/Desktop/研究生课内资料/语音处理/实验3/dpss-exp3-VC-BNF# python preprocess.py --data_dir dataset/ --save_dir save_
data/exp3-data-all/
Set up BNFs extraction network
Loading BNFs extractor from ./config/asr_config.yaml
read 351 lines from ./pretrained_model/asr_model/dict.txt
example(last) <sos/eos> 350

Extracting mel-spectrograms, spectrograms and f0s...
init praat-f0 extractor with 20/600
5997it [46:17,  2.16it/s]
Done extracting features!
[]
```

4.2 根据说话人嵌入网络的有关原理，确定说话人嵌入网络的某种特定结构（鼓励探索更好的模型结构），并实现 `models/models.py` 中 SPKEmbedding 类的具体代码；如果有需要，可进一步对 `models/models.py` 中 BLSTMToManyConversionModel 类的相应部分进行修改。

实现SPKEmbedding 类的具体代码如下：

```python
class SPKEmbedding(nn.Module):
    def __init__(self, num_spk, embd_dim):
        super(SPKEmbedding, self).__init__()
        # define your module components below
        # e.g. self.embedding_table = ...
        #from zxt
        self.embedding_table = nn.Embedding(num_spk, embd_dim)
        layers = []
        layers.append(nn.Linear(embd_dim, embd_dim))
        layers.append(nn.ReLU())
        layers.append(nn.Linear(embd_dim, embd_dim))
        self.layers = nn.Sequential(*layers)

    def forward(self, spk_inds):
        # define your inference process below
        # e.g. return self.embedding_table(spk_inds)
        x = self.embedding_table(spk_inds)
        return self.layers(x)
```

修改后的BLSTMToManyConversionModel类代码如下：

```
class BLSTMToManyConversionModel(nn.Module):
    def __init__(self, in_channels, out_channels, num_spk, embd_dim, lstm_hidden):
        super(BLSTMToManyConversionModel, self).__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.num_spk = num_spk
        self.embd_dim = embd_dim
        self.lstm_hidden = lstm_hidden
        self.spk_embed_net = SPKEmbedding(num_spk=num_spk,
                                          embd_dim=embd_dim)
        self.emb_proj1 = nn.Linear(in_features=embd_dim,
                                   out_features=in_channels)
        self.emb_proj2 = nn.Linear(in_features=embd_dim,
                                   out_features=lstm_hidden * 2)
        self.blstm1 = nn.LSTM(input_size=in_channels,
                              hidden_size=lstm_hidden,
                              bidirectional=True)
        self.blstm2 = nn.LSTM(input_size=lstm_hidden * 2,
                              hidden_size=lstm_hidden,
                              bidirectional=True)
        self.out_projection = nn.Linear(in_features=2 * lstm_hidden,
                                        out_features=out_channels)

    def forward(self, x, spk_inds):
        # look up speaker embedding
        spk_embds = self.spk_embed_net(spk_inds)
        spk_embds =spk_embds.repeat(x.shape[0],1,1)

        # add speaker embd to the inputs
        blstm1_inputs = x + self.emb_proj1(spk_embds) # give your implementation here
        # pass to the 1st BLSTM layer
        blstm1_outs, _ = self.blstm1(blstm1_inputs)
        # add speaker embd to the outputs of 1st lstm
        blstm2_inputs = blstm1_outs + self.emb_proj2(spk_embds)  # give your implementation here
        # pass to the 2nd BLSTM layer
        blstm2_outs, _ = self.blstm2(blstm2_inputs)
        # project to the output dimension
        outputs = self.out_projection(blstm2_outs)
        return outputs
```

4.3 在实验报告中说明所实现的 SPKEmbedding 的具体结构，并给出网络的基本结构图。

(1)

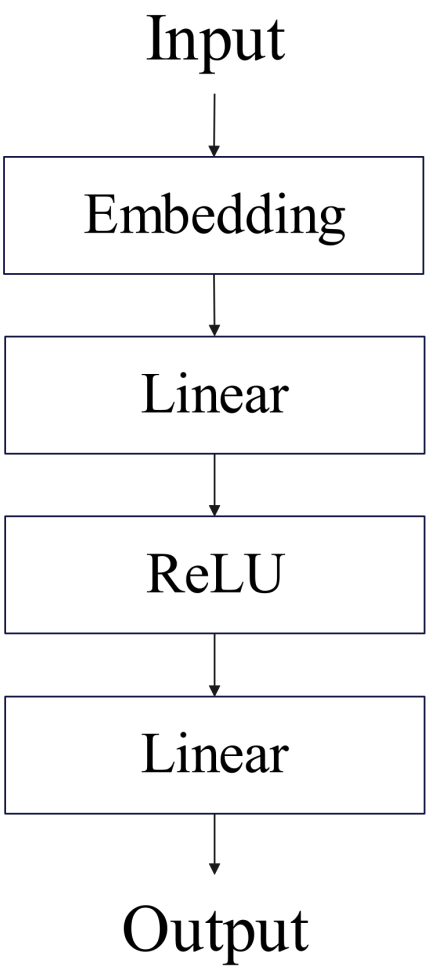
实现的 SPKEmbedding 具体结构如下：

- **嵌入查找表**：使用 `nn.Embedding(num_spk, embd_dim)` ，将每个说话人索引映射到一个固定维度的嵌入空间中。该查找表的目的是将每个说话人的索引映射为固定的低维向量表示（即说话人嵌入）。
- **全连接层1**：使用 `nn.Linear(embd_dim, embd_dim)` ，将嵌入向量维度映射为相同维度 `embd_dim` 。
- **激活函数**：使用 `nn.ReLU()` ，在映射后增加非线性变换。ReLU 的非线性特性使得网络能够学习到更复杂的特征。
- **全连接层2**：使用 `nn.Linear(embd_dim, embd_dim)` ，将结果再映射回 `embd_dim` 维度。

“全连接层+激活函数+全连接层”部分可以看作是对说话人嵌入进行进一步加工，提取出更丰富的特征，增加模型的表达能力。

(2)

网络基本结构图如下：



4.4 若对 BLSTMToManyConversionModel 类进行了修改，也需要在实验报告中加以说明。

我只完成了两个填空，没有进行其它修改：

```
# add speaker embd to the inputs
blstm1_inputs = x + self.emb_proj1(spk_embds)


# add speaker embd to the outputs of 1st lstm
blstm2_inputs = blstm1_outs + self.emb_proj2(spk_embds)
```

4.5 根据 GitHub repository 的说明文档中的要求（Any-to-Many → Train），运行相关命令进行模型训练。

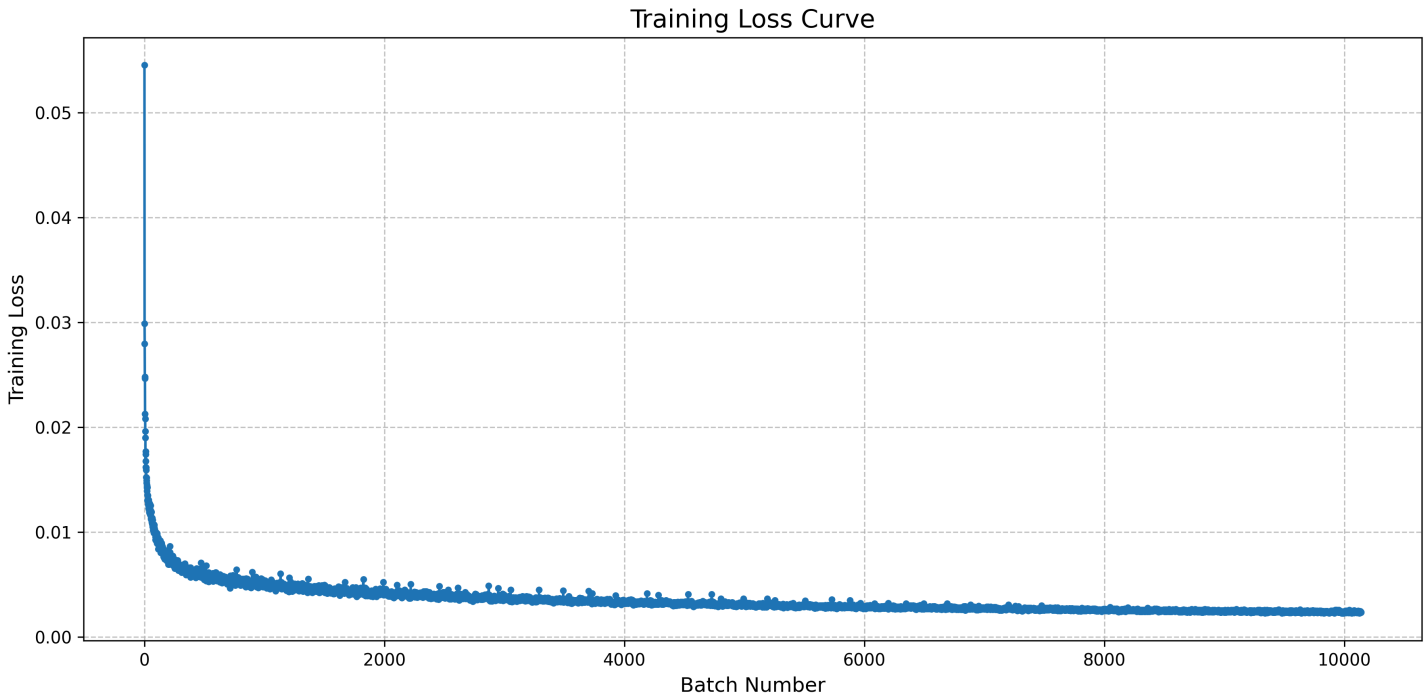
使用的训练指令为

```
CUDA_VISIBLE_DEVICES=6 nohup python train_to_many.py \
--model_dir  pretrained_model/asr_model/many/ \
--test_dir result/many/ \
--data_dir save_data/exp3-data-all/ > run_4.log 2>&1 &
```

4.6 给出模型训练时的损失函数曲线；模型训练了多少个 epoch？训练完成后，最终（对应最后一个 epoch）的平均训练 MSE loss 是多少?

(1)

模型训练时的损失函数曲线如下：



(2)

模型训练了60个epoch。

(3)

最终的平均训练 MSE loss 是0.00240。

4.7 基于上述训练好的模型，在验证集上进行验证。在验证集上的 MSE loss 是多少?

根据训练日志 `run_4.log`， 验证集上的 MSE loss 是0.00468。

4.8 基于上述训练好的模型，在测试集上进行测试。针对某句测试用例，通过给定不同的目标说话人的 Speaker ID，得到同一个源说话人的语音对应的不同目标说话人的转换语音；听辨转换语音的音色与目标说话人的音色，是否存在差异？为什么?

(1)

以 `bzn/00001.wave` 为测试用例，得到 `mst-female` 和 `mst-male` 说话人的转换语音。

(2)

对 `mst-female` ， 转换语音的音色与目标说话人的音色几乎没有差异，声音很清晰，没有噪声。

对 `mst-male` ， 转换语音的音色与目标说话人的音色没有明显差异，不过存在一定噪声，类似于回音、颤音。

4.9 根据 GitHub repository 的说明文档中的要求（Any-to-Many → Inference），将某个特定源说话人的语音（src_wav 参数）转换为某个特定目标说话人（tgt_spk 参数）的语音。

使用的推理指令为：

```
python inference_to_many.py \
--src_wav dataset/bzn/000001.wav \
--tgt_spk mst-female \
--ckpt pretrained_model/asr_model/many/bnf-vc-to-many-59.pt \
--save_dir result/many_inference/
```

```
python inference_to_many.py \
--src_wav dataset/bzn/000001.wav \
--tgt_spk mst-male \
--ckpt pretrained_model/asr_model/many/bnf-vc-to-many-59.pt \
--save_dir result/many_inference/
```

4.10 自己录制一段语音，并作为模型的输入（src_wav 参数），对提供的中的 3 个目标说话人（改变 tgt_spk 参数）进行语音转换，得到 3 个转换后的语音。

录制了 `record.wav` 。
然后使用下述3个推理命令，即可得到3个转换后的语音：

```
python inference_to_many.py \
--src_wav record.wav \
--tgt_spk bzn \
--ckpt pretrained_model/asr_model/many/bnf-vc-to-many-59.pt \
--save_dir result/many_inference/
```

```
python inference_to_many.py \
--src_wav record.wav \
--tgt_spk mst-female \
--ckpt pretrained_model/asr_model/many/bnf-vc-to-many-59.pt \
--save_dir result/many_inference/
```

```
python inference_to_many.py \
--src_wav record.wav \
--tgt_spk mst-male \
--ckpt pretrained_model/asr_model/many/bnf-vc-to-many-59.pt \
--save_dir result/many_inference/
```

5 任务五：第 16 周实验课“惊喜”任务

5.1 语音转换：请根据 FTP 中提供的源音频，调用自己训练好的模型进行语音转换，转换后的音频上传至 FTP 同目录。对于 input1.wav 请将其转换为 bzn 的音色，并存储为 output1.wav 上传至 FTP；对于 input2.wav 请将其转换为 mst_male 的音色，并存储为 output2.wav 上传至 FTP；对于 input3.wav 请将其转换为 mst_female 的音色，并存储为 output3.wav 上传至 FTP。

将FTP上的 `input1.wav`，`input2.wav`，`input3.wav` 下载到 `dpss-exp3-VC-BNF` 文件夹下，然后分别运行以下3个命令：

```
python inference_to_many.py \
--src_wav input1.wav \
--tgt_spk bzn \
--ckpt pretrained_model/asr_model/many/bnf-vc-to-many-59.pt \
--save_dir result/class/
```

```
python inference_to_many.py \
--src_wav input2.wav \
--tgt_spk mst-male \
--ckpt pretrained_model/asr_model/many/bnf-vc-to-many-59.pt \
--save_dir result/class/
```

```
python inference_to_many.py \
--src_wav input3.wav \
--tgt_spk mst-female \
--ckpt pretrained_model/asr_model/many/bnf-vc-to-many-59.pt \
--save_dir result/class/
```

再将生成的 `input1-to-bzn-converted-59.wav` 改名为 `output1.wav`， `input2-to-mst-male-converted-59.wav` 改名为 `output2.wav`， `input3-to-mst-female-converted-59.wav` 改名为 `output3.wav`， 上传FTP即可。

5.2 主观测评：请对 FTP 中提供的音频的转换效果进行评分，并整理成符合格式的 result.txt 上传至 FTP 同目录。评分项：从转换音频与目标音色的相似度和音频内容的可懂度两方面分别进行打分。评分制： 5 分制，最低为 1 分，最高为 5 分。

完成评分后得到的 `result.txt` 文件如下：

```
33d4a957a6f281fbc83f62533c27b37e 4 5
60c1b8db5a52739d04c0ec00d416c4db 4 5
86f5b90f20377ab68bb6b0b5d0ea7fd6 5 5
261ee44c1ee03e8e83bb3ad32383d487 3 3
273c4bc27354b9b903eb0bac2251f642 5 3
0340e5688e940169d26a12ab3330f9cd 4 5
912f33d05331e596148a8414e3c30acc 4 4
61587a203455c8134aa46c425240f922 3 4
74825f97f2ec91250547534294576b35 3 4
443446e88ca8af15f85092b1a3dd4405 5 5
a7a3293382ec745b9d4b339bc73315fd 4 4
ebbfac91f27bda11e31e1584df7e3c10 4 3
```