实验 7-1 模型推理: Tacotron2 语音合成

智能计算系统课程设计实验报告

班级：110

姓名：耿翊中

学号：2021213382

2023 年 11 月

目录

1. 实验背景

前述章节实验完成了图像风格迁移应用在深度学习处理器上的迁移、开发和优化。本章实验将涉及多个不同领域(如目标检测、语音合成、SQuAD 问答、图像分类等)的人工智能应用在深度学习处理器上的开发和优化。

第 7.1 节介绍 DLP 平台上实现基于 Tacotron2 模型的语音合成任务。主要的实验内容是掌握 Tacotron2 的算法细节，包括位置敏感注意力模块、Encoder 模块、Decoder 模块 Tacotron2 模块以及推理模块的实现方法，并能够在 DLP 平台上进行推理实现。

语音合成(一般指文本到语音合成 Text-To-Speech Synthesis，TTS)旨在通过给定文本生成清晰自然的语音，一直以来都是人工智能领域的热门研究方向。经典的语音合成算法通常依次由以下三部分组成:

1)文本分析:对文本进行归一化、分词等处理;

2)声学模型(Acoustic Model): 将输的文本信息转换为声学特征(通常是 Mel 频谱)既可以使用传统的概率模型(如 HMM)，也可使用基于 CNN/RNN/self-attention 等构建的网络模型;

3)声码器(Vocoder):基于声学特征生成语音音频作为最终输出。这一部分既可以使用传统基于参数合成的声码器，也可使用 WaveNet、WaveGlow、WaveGAN、WaveVAE 等多种基于神经网络的声码器。

较早的研究者通常使用参数统计方法合成语音，此后声码器和声学模型逐步被深度模型取代(WaveNet、DeepVoice 系列、Tacotron 系列、FastSpeech 系列、AdaSpeech 系列等)。目前则出现了一些完全端到端的语音合成模型,典型的有 Tacotron/Tacotron2 模型、deepvoice3 模型、FastSpeech/FastSpeech2 模型、VITS 模型等。本实验将要关注的语音合成算法就是 Tacotron2.

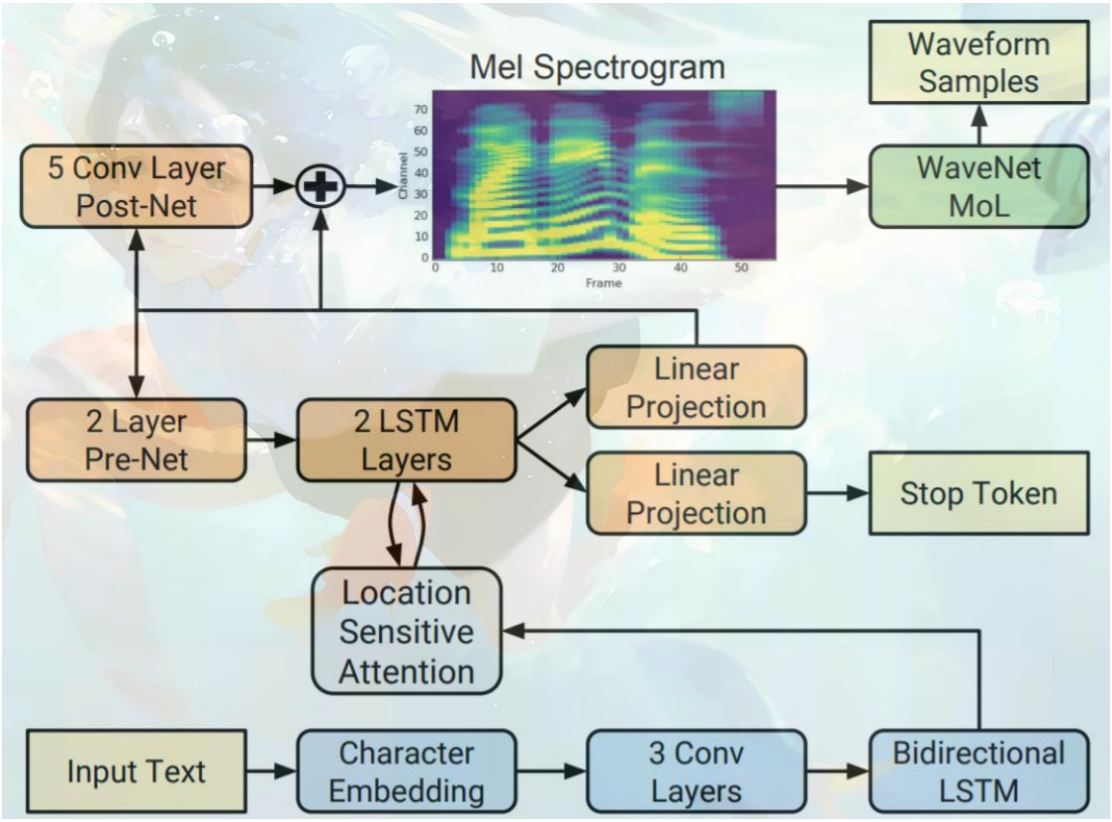
2. 实验原理

2.1 模型生成原理

Tacotron 2 是一种先进的语音合成系统，它将语音合成过程分为两个主要部分：第一部分是循环 seq2seq 结构的特征预测网络，负责把字符向量映射为梅尔声谱图，第一部分后再接一个 WaveNet 模型的修订版本，即 WaveGlow，负责把梅尔声谱图转换为固定大小的特征表示。然后，在自回归解码器使用该特征表示，每次生成一个频谱图帧。

2.2 Tacotron2 结构原理

Tacotron2 网络结构如下：



针对上图结构，可具体分析 Tacotron2 模型的结构：

1. 输入文本和字符嵌入（Input Text & Character Embedding）：系统接收文本输入，每个字符通过一个嵌入层（embedding layer）转换成一个高维向量，这个向量能够代表每个字符的独特属性。

2. 三个卷积层（3 Conv Layers）：字符嵌入通过三个卷积神经网络层，这些层可以帮助模型捕捉字符序列中的局部相关性。

3. 双向长短期记忆网络（Bidirectional LSTM）：卷积层的输出被送到双向长短期记忆网络。这是一种循环神经网络（RNN），能够处理序列数据，并捕获长距离的依赖关系。

4. 位置敏感的注意力机制（Location Sensitive Attention）：注意力机制帮助模型在生

成声音的每一步集中于输入文本的不同部分。

5. 预测网络（Pre-Net）和两个 LSTM 层（2 LSTM Layers）：预测网络是处理注意力机制输出的前置网络，之后是两个 LSTM 层，它们进一步处理序列信息。

6. 线性投影（Linear Projection）: LSTM 层的输出通过线性层，这些层的目的是预测梅尔频谱图（Mel Spectrogram），这是一种音频的表示方式，可以捕捉到声音的基本特征。

7. 后处理网络（Post-Net）：后处理网络进一步改善梅尔频谱图的质量。

8. 梅尔频谱图（Mel Spectrogram）：系统的输出，梅尔频谱图是一种表示音频信息的图表，它显示了不同频率随时间变化的强度。

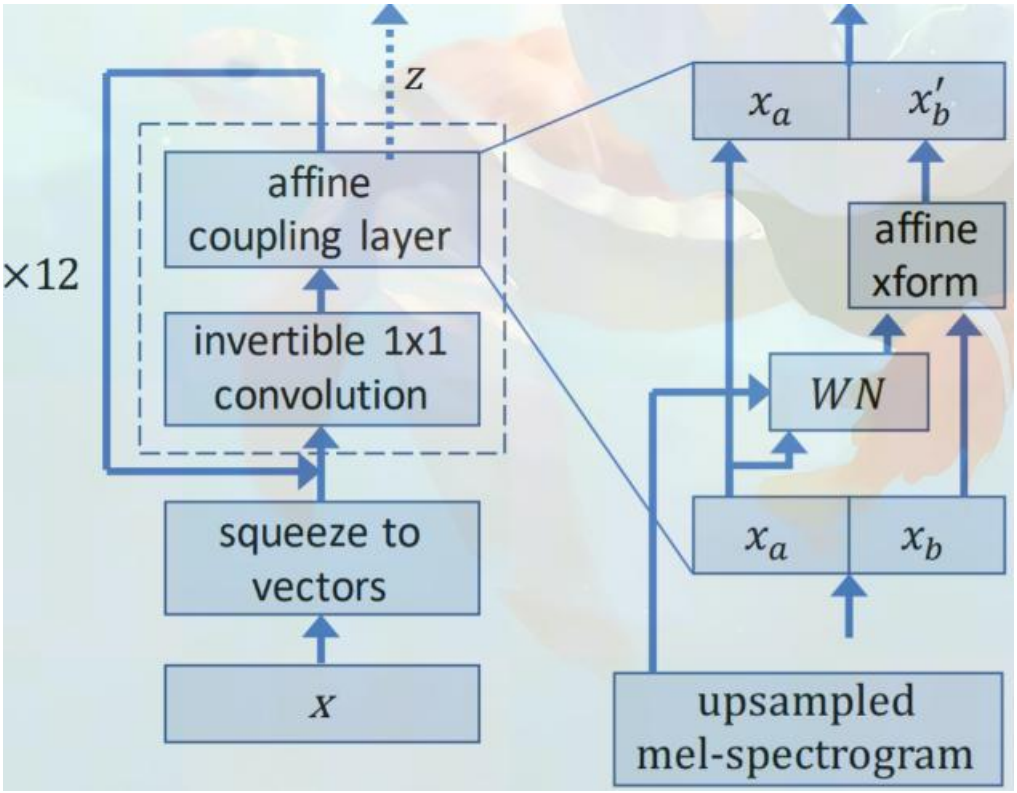
9. WaveNet 声码器（WaveNet MoL）：最终，梅尔频谱图被送入 WaveNet 模型，它是一个深度神经网络，能够从频谱图生成最终的波形样本。

10. 波形样本（Waveform Samples）：这是最终的输出，代表合成的语音波形。

11. 停止标记（Stop Token）：用于预测文本输入结束的信号，以便确定合成语音的结束时机。

2.3 WaveGlow 原理

Waveglow 结构如下：



WaveGlow 结合了 Glow（一种基于流的生成模型）的特点和 WaveNet（一个自回归的深度神经网络）的语音合成能力。它的目标是从梅尔频谱图生成语音波形。

WaveGlow 模型的主要步骤和组件包括：

输入（x）：模型接收一个经过上采样的梅尔频谱图作为输入。

压缩到向量（Squeeze to vectors）：输入首先被压缩到向量，这是为了减少维度并为后续的流操作做准备。

可逆 1x1 卷积（Invertible 1x1 convolution）：这一步骤使用 1x1 卷积对向量进行变换，这是一个可逆的过程，意味着从输出可以完全恢复输入，这对生成模型的可逆性质是非常重要的。

仿射耦合层（Affine coupling layer）：这是 WaveGlow 模型的核心，它分别处理输入向量

的两部分，然后通过特定的函数（通常是神经网络）调整它们的尺度和平移，这样可以改变数据的分布而不损失信息。

重复多次：上述的可逆卷积和仿射耦合层会被重复多次（如图中的×12），以便逐步地改变和细化数据的分布。

最终输出（z）：经过多次变换后，模型输出一个隐变量（z），它遵循一个简单的分布（如高斯分布），并可以被转化为音频波形。

WaveGlow 的作用是提供一种从给定梅尔频谱图到生成可听语音波形的高效路径。它被设计为无需自回归的过程，因此生成速度快，适合实时的语音合成应用。由于它的这些属性，WaveGlow 成为了当代语音合成领域的一个重要工具，特别是在需要生成自然 sounding 语音的场合。

3. 实验流程

本次实验主要步骤如下图所示：

3.1 预处理

预处理部分包括设定超参数，文本前处理和生成梅尔声谱图

3.1.1 设定超参数

首先需要设定相关参数的值，如与训练模型的路径、推理设备、输入文本长度等。通过下列代码进行解析命令行传过来的参数

```
import sys

cur_path = os.getcwd()
models_path = cur_path + "/src/"
sys.path.append(models_path)

```

```
from inference import checkpoint_from_distributed, unwrap_distributed, MeasureTime, prepare_input_sequence,load_and_setup_model
import models
import torch
import argparse
import numpy as np
from scipy.io.wavfile import write

import time
import dllogger as DLLogger
from dllogger import StdOutBackend, JSONStreamBackend, Verbosity
# cambricon-note: no apex in cambricon-pytorch
#from apex import amp
from waveglow.denoiser import Denoiser

# cambricon-note: use cambricon-pytorch
import torch_mlu


def parse_args(parser):

    """
    Parse commandline arguments.
    """
    parser.add_argument('--tacotron2', type=str,
                        help='full path to the Tacotron2 model checkpoint file')
    parser.add_argument('--waveglow', type=str,
                        help='full path to the WaveGlow model checkpoint file')
    parser.add_argument('-s', '--sigma-infer', default=0.6, type=float)
    parser.add_argument('-d', '--denoising-strength', default=0.01, type=float)
    parser.add_argument('-sr', '--sampling-rate', default=22050, type=int,
                        help='Sampling rate')
    parser.add_argument('--amp-run', action='store_true',
                        help='inference with AMP')
    parser.add_argument('--log-file', type=str, default='nvlog.json',
                        help='Filename for logging')
    parser.add_argument('--stft-hop-length', type=int, default=256,
                        help='STFT hop length for estimating audio length from mel size')
    parser.add_argument('--num-iters', type=int, default=10,
                        help='Number of iterations')
    parser.add_argument('-il', '--input-length', type=int, default=64,
                        help='Input length')
    parser.add_argument('-bs', '--batch-size', type=int, default=1,
                        help='Batch size')
```

```
    parser.add_argument('--device-param', type=str, default='mlu',
                        help='inference device, optional val is gpu/mlu/cpu.')
    return parser


#------------ main -------------------------------
parser = argparse.ArgumentParser(description='PyTorch Tacotron 2 Inference')
parser = parse_args(parser)
cur_path = os.getcwd()
pretrained_model_dir = cur_path + "/../../model/pretrained/pytorch_tacotron2_inference/"

args, unknown_args = parser.parse_known_args()
# 设定预训练模型路径，推理设备，waveglow_channel，输入文本的 input_length
args.tacotron2 = pretrained_model_dir + "nvidia_tacotron2pyt_fp32_20190427"
args.waveglow = pretrained_model_dir + "nvidia_waveglowpyt_fp32_20190427"
args.device_param = "mlu"
args.input_length=166  # 待合成的文本长度，下面例子中一句话长度是 166
args.batch_size=1
args.num_iters=14
# wn_channels 参数的解析在 models.py 完成
args.wn_channels=512

print("args:",args,unknown_args)
```

--tacotron2: 指定 Tacotron 2 模型检查点文件的完整路径。

--waveglow: 指定 WaveGlow 模型检查点文件的完整路径。

-s 或 --sigma-infer: 设置在推理过程中用于 WaveGlow 模型的 sigma 参数，默认值为 0.6。

-d 或 --denoising-strength: 设置去噪强度，默认值为 0.01。

-sr 或 --sampling-rate: 设置采样率，默认值为 22050 Hz。

--amp-run: 如果提供此参数，将使用自动混合精度（AMP）进行推理。

--log-file: 日志文件的名称，默认为'nvlog.json'。

--stft-hop-length: 用于从 mel 大小估计音频长度的 STFT 跳跃长度，默认值为 256。

--num-iters: 推理的迭代次数，默认值为 10。

-il 或 --input-length: 输入长度，默认值为 64。

-bs 或 --batch-size: 批处理大小，默认值为 1。

--device-param: 推理设备，可选的值为 gpu/mlu/cpu，默认值为'mlu'。

在代码的后半部分，这些参数中的一些被直接设置了新的值：

args.tacotron2 被设置为预训练的 Tacotron 2 模型的路径。

args.waveglow 被设置为预训练的 WaveGlow 模型的路径。

args.device_param 被设置为使用'mlu'设备进行推理。

args.input_length 被设置为 166，这是指定的待合成文本的长度。

args.batch_size 被设置为 1，表示每批处理的样本数。

args.num_iters 被设置为 14，表示将执行的推理迭代次数。

args.wn_channels 被设置为 512，这个参数是在其他地方（如 models.py）解析的，通常与 WaveGlow 模型的通道数有关。

这些参数对于配置模型的推理行为至关重要，包括模型的位置、推理设备、去噪参数、迭代次数等。通过这些参数，用户可以根据特定的需求或硬件配置来定制推理过程。

3.1.2 文本前处理

```
    # cambricon-note: use native tacotron2 model to infer
        # jitted_tacotron2 = torch.jit.script(tacotron2)
        texts = [
            "The forms of printed letters should be beautiful, and that their arrangement on the page should be reasonable and a help to the shapeliness of the letters themselves. The forms of printed letters should be beautiful, and that their arrangement on the page should be reasonable and a help to the shapeliness of the letters themselves."]
        print(len(texts))
        texts = [texts[0][:args.input_length]]
        texts = texts * args.batch_size
```

首先，定义一个文本列表 texts，这个列表包含了将要被转换为语音的文本样本。这里只有一个很长的文本样本，它包含了关于打印字母形式和排版的描述。

使用 print(len(texts)) 输出文本列表的长度，此时长度为 1，因为列表中只有一个文本样本。

接下来，根据 args.input_length 参数截断文本。texts[0][:args.input_length] 表示取第一个（也是唯一一个）文本样本的前 args.input_length 个字符。这是因为模型可能对输入长度有限制，过长的文本需要被截断以适应模型的要求。

然后，文本被复制 args.batch_size 次，以创建一个批处理列表。texts * args.batch_size 会生成一个新的列表，其中包含 args.batch_size 个重复的文本字符串。

这个前处理步骤确保了无论模型期望的输入长度或批处理大小如何，都能提供正确形式的输入。如果 args.batch_size 大于 1，这意味着在一个批次中可以处理多个相同的文本样本，这通常用于性能测试，以确保可以连续处理多个请求而不仅仅是单个请求。在实际应用中，这些文本样本可能是不同的，以便同时处理多个转换请求。

3.1.3 加载模型

```
    tacotron2 = load_and_setup_model('Tacotron2', parser, args.tacotron2, args.amp_run, args.device_param, forward_is_infer=True)
    waveglow = load_and_setup_model('WaveGlow', parser, args.waveglow, args.amp_run, args.device_param)


    if args.device_param == "cpu":
        denoiser = Denoiser(waveglow, args.device_param)
    elif args.device_param == "mlu":
        denoiser = Denoiser(waveglow, args.device_param).mlu()
    else:
        denoiser = Denoiser(waveglow, args.device_param).cuda()

```

完成超惨设定和文本前处理后，还需要根据传入的命令行参数加载预处理模型，并将模型拷贝到 MLU 上来。两个模型分别是 Tacotron2 和 waveglow。

3.2 预测生成音频文件

3.2.1 文本特征序列生成梅尔频谱图

在对输入文本完成前处理得到文本序列特征 sequences_paded 后，即可开始基于 sequences_padded 生成梅尔声谱图。

```
    for iter in range(args.num_iters):

        measurements = {}

        with MeasureTime(measurements, "pre_processing", args.device_param):
            sequences_padded, input_lengths = prepare_input_sequence(texts, args.device_param)
        # TODO: 禁用梯度计算，进行推理过程。
        with torch.no_grad():
            with MeasureTime(measurements, "latency", args.device_param):
                with MeasureTime(measurements, "tacotron2_latency", args.device_param):
                    # mel, mel_lengths, _ = tacotron2(sequences_padded, input_lengths)
                    mel, mel_lengths, alignment = tacotron2(sequences_padded, input_lengths)
```

生成梅尔频谱图：

在不计算梯度的情况下（使用 torch.no_grad()上下文管理器），Tacotron 2 模型接收文本特征序列和它们的长度。

模型通过其内部结构，通常包括编码器、注意力机制和解码器，处理这些输入序列。

编码器首先将输入的文本特征序列转换为中间特征表示。

注意力机制决定模型在生成每个音频帧时应该集中在输入文本的哪个部分。

解码器逐步生成梅尔频谱图，每一步都基于编码器的输出和注意力机制的指导。

Tacotron 2 模型的推理：

在推理循环中（for iter in range(args.num_iters):），模型对每个文本序列调用一次。

在 MeasureTime 的上下文中，调用 Tacotron 2 模型的前向函数（tacotron2(sequences_padded, input_lengths)），其中 sequences_padded 是填充后的文本特征序列，input_lengths 是序列的实际长度。

Tacotron 2 模型的输出是梅尔频谱图和对应的长度，以及可选的对齐矩阵（显示输入文本和输出频谱之间的对齐情况）。

梅尔频谱图的输出：

推理得到的梅尔频谱图就是 Tacotron 2 模型对输入文本序列的直接输出。这些频谱图包含了可以转换为音频波形的信息。

3.2.2 从梅尔频谱图生成语音

得到 mel 谱后，便利用 WaveGlow 生成 audio 数据，并将 audio 数据保存到本地。在本实验中，将直接展示合成的 audio 音频，运行下方代码块后，即可得到支持在线播放的 audio 音频。

```
with MeasureTime(measurements, "waveglow_latency", args.device_param):
                    audios = waveglow.infer(mel, sigma=args.sigma_infer)
                    audios = audios.float()
                    audios = denoiser(audios, stren
```

之后，还可以选择移除 waveglow 中的 bias 使得生成音频更加清晰

```
audios_denoiser = denoiser(audios_float, strength=args.denoising_strength).squeeze(1)
audios_denoiser_npy = audios_denoiser.cpu().numpy()
audios_npy = audios_float.cpu().numpy()
ipd.Audio(audios_denoiser_npy, rate=args.sampling_rate)
```

4. 实验结果及分析

最终评测输出结果：

```
得分：300.00   最后一次提交时间：2024-01-09 16:22:52
Accept


TEST_ATTEN  ENCODER DECODER TACOTRON2   AUDIO
PASS
PASS
PASS
PASS
PASS
Attention Context: tensor([[-0.7686, 0.4479, -0.3848, -0.3220]], grad_fn=)
Attention Weights: tensor([[0.1249, 0.1688, 0.2372, 0.2462, 0.2229]], grad_fn=)
Context Matches Expected: True
Weights Matches Expected: True
********************
Pass!
********************
[2024-1-9 16:46:1] [CNNL] [Warning]: [cnnlGetConvolutionForwardAlgorithm] is deprecated and will b
e removed in the future release. See cnnlFindConvolutionForwardAlgorithm() API for replacement.
[2024-1-9 16:46:7] [CNNL] [Warning]: [cnnlClip] is deprecated and will be removed in the future re
lease, please use [cnnlClip_v2] instead.
args: Namespace(amp_run=False, batch_size=1, denoising_strength=0.01, device_param='mlu', input_le
ngth=167, log_file='nvlog_bs1_il128_fp32.json', num_iters=14, sampling_rate=22050, sigma_infer=0.6,
stft_hop_length=256, tacotron2='/cg/tacotron2_inference/model/pretrained/nvidia_tacotron2pyt_fp32_20
190427', waveglow='/cg/tacotron2_inference/model/pretrained/nvidia_waveglowpyt_fp32_20190427', wn_ch
annels=512) ['--wn-channels', '512']
DLL 2024-01-09 16:45:53.491755 - PARAMETER tacotron2 : /cg/tacotron2_inference/model/pretrained/nv
idia_tacotron2pyt_fp32_20190427
DLL 2024-01-09 16:45:53.491851 - PARAMETER waveglow : /cg/tacotron2_inference/model/pretrained/nvi
dia_waveglowpyt_fp32_20190427
```

```
DLL 2024-01-09 16:45:53.491896 - PARAMETER sigma_infer : 0.6
DLL 2024-01-09 16:45:53.491937 - PARAMETER denoising_strength : 0.01
DLL 2024-01-09 16:45:53.491975 - PARAMETER sampling_rate : 22050
DLL 2024-01-09 16:45:53.492011 - PARAMETER amp_run : False
DLL 2024-01-09 16:45:53.492048 - PARAMETER log_file : nvlog_bs1_il128_fp32.json
DLL 2024-01-09 16:45:53.492082 - PARAMETER stft_hop_length : 256
DLL 2024-01-09 16:45:53.492115 - PARAMETER num_iters : 14
DLL 2024-01-09 16:45:53.492148 - PARAMETER input_length : 167
DLL 2024-01-09 16:45:53.492181 - PARAMETER batch_size : 1
DLL 2024-01-09 16:45:53.492215 - PARAMETER device_param : mlu
DLL 2024-01-09 16:45:53.492247 - PARAMETER wn_channels : 512
DLL 2024-01-09 16:45:53.492280 - PARAMETER model_name : Tacotron2_PyT
args: Namespace(amp_run=False, batch_size=1, denoising_strength=0.01, device_param='mlu', input_length=167, log_file='nvlog_bs1_il128_fp32.json', num_iters=14, sampling_rate=22050, sigma_infer=0.6, stft_hop_length=256, tacotron2='/cg/tacotron2_inference/model/pretrained/nvidia_tacotron2pyt_fp32_20190427', waveglow='/cg/tacotron2_inference/model/pretrained/nvidia_waveglowpyt_fp32_20190427', wn_channels=512) ['--wn-channels', '512']
1
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:46:35.971860 - 0 pre_processing : 0.0031085191294550896
DLL 2024-01-09 16:46:35.972042 - 0 tacotron2_latency : 1.8157252855598927
DLL 2024-01-09 16:46:35.972094 - 0 waveglow_latency : 6.336324874311686
DLL 2024-01-09 16:46:35.972137 - 0 latency : 8.152102591469884
DLL 2024-01-09 16:46:35.972179 - 0 type_conversion : 1.891050487756729e-05
DLL 2024-01-09 16:46:35.972222 - 0 data_transfer : 0.0006193723529577255
DLL 2024-01-09 16:46:35.972261 - 0 storage : 0.002255616709589958
DLL 2024-01-09 16:46:35.972299 - 0 tacotron2_items_per_sec : 408.1013828980781
DLL 2024-01-09 16:46:35.972341 - 0 waveglow_items_per_sec : 29937.858894996552
DLL 2024-01-09 16:46:35.972377 - 0 num_mels_per_audio : 741
DLL 2024-01-09 16:46:35.972413 - 0 throughput : 23269.57958042533
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
```

```
DLL 2024-01-09 16:46:44.852760 - 1 pre_processing : 0.003173358738422394
DLL 2024-01-09 16:46:44.852878 - 1 tacotron2_latency : 1.969293912872672
DLL 2024-01-09 16:46:44.852925 - 1 waveglow_latency : 6.9032291024923325
DLL 2024-01-09 16:46:44.852966 - 1 latency : 8.872575175017118
DLL 2024-01-09 16:46:44.853006 - 1 type_conversion : 1.893937587738037e-05
DLL 2024-01-09 16:46:44.853047 - 1 data_transfer : 0.0027780774980783463
DLL 2024-01-09 16:46:44.853084 - 1 storage : 0.0016326447948813438
DLL 2024-01-09 16:46:44.853125 - 1 tacotron2_items_per_sec : 405.22138152345775
DLL 2024-01-09 16:46:44.853163 - 1 waveglow_items_per_sec : 29593.107365688345
DLL 2024-01-09 16:46:44.853198 - 1 num_mels_per_audio : 798
DLL 2024-01-09 16:46:44.853231 - 1 throughput : 23024.656987434977
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:46:53.060831 - 2 pre_processing : 0.003199509344995022
DLL 2024-01-09 16:46:53.061012 - 2 tacotron2_latency : 1.809855449013412
DLL 2024-01-09 16:46:53.061061 - 2 waveglow_latency : 6.391275024972856
DLL 2024-01-09 16:46:53.061105 - 2 latency : 8.201185974292457
DLL 2024-01-09 16:46:53.061146 - 2 type_conversion : 2.0209699869155884e-05
DLL 2024-01-09 16:46:53.061188 - 2 data_transfer : 0.001223534345626831
DLL 2024-01-09 16:46:53.061226 - 2 storage : 0.0017671054229140282
DLL 2024-01-09 16:46:53.061262 - 2 tacotron2_items_per_sec : 405.5572506633709
DLL 2024-01-09 16:46:53.061297 - 2 waveglow_items_per_sec : 29400.080463725317
DLL 2024-01-09 16:46:53.061395 - 2 num_mels_per_audio : 734
DLL 2024-01-09 16:46:53.061435 - 2 throughput : 22911.80819323038
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:47:01.343301 - 3 pre_processing : 0.003196178935468197
DLL 2024-01-09 16:47:01.343435 - 3 tacotron2_latency : 1.8109513130038977
DLL 2024-01-09 16:47:01.343483 - 3 waveglow_latency : 6.464665078558028
DLL 2024-01-09 16:47:01.343525 - 3 latency : 8.275674780830741
DLL 2024-01-09 16:47:01.343565 - 3 type_conversion : 1.9179657101631165e-05
DLL 2024-01-09 16:47:01.343622 - 3 data_transfer : 0.0007441630586981773
DLL 2024-01-09 16:47:01.343661 - 3 storage : 0.0020543355494737625
DLL 2024-01-09 16:47:01.343697 - 3 tacotron2_items_per_sec : 406.9684230094008
DLL 2024-01-09 16:47:01.343733 - 3 waveglow_items_per_sec : 29185.115966144393
DLL 2024-01-09 16:47:01.343767 - 3 num_mels_per_audio : 737
DLL 2024-01-09 16:47:01.343801 - 3 throughput : 22798.382608875363
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:47:09.477562 - 4 pre_processing : 0.0032207798212766647
DLL 2024-01-09 16:47:09.477701 - 4 tacotron2_latency : 1.8286345237866044
```

```
DLL 2024-01-09 16:47:09.477751 - 4 waveglow_latency : 6.298166464082897
DLL 2024-01-09 16:47:09.477794 - 4 latency : 8.126857277937233
DLL 2024-01-09 16:47:09.477833 - 4 type_conversion : 1.9670464098453522e-05
DLL 2024-01-09 16:47:09.477874 - 4 data_transfer : 0.0014728056266903877
DLL 2024-01-09 16:47:09.477911 - 4 storage : 0.0020003048703074455
DLL 2024-01-09 16:47:09.477947 - 4 tacotron2_items_per_sec : 401.93925600727204
DLL 2024-01-09 16:47:09.477982 - 4 waveglow_items_per_sec : 29875.36151561513
DLL 2024-01-09 16:47:09.478017 - 4 num_mels_per_audio : 735
DLL 2024-01-09 16:47:09.478051 - 4 throughput : 23152.861378631096
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:47:17.762066 - 5 pre_processing : 0.003199789673089981
DLL 2024-01-09 16:47:17.762180 - 5 tacotron2_latency : 1.839544976130128
DLL 2024-01-09 16:47:17.762227 - 5 waveglow_latency : 6.438214439898729
DLL 2024-01-09 16:47:17.762268 - 5 latency : 8.277811606414616
DLL 2024-01-09 16:47:17.762331 - 5 type_conversion : 1.8999911844730377e-05
DLL 2024-01-09 16:47:17.762374 - 5 data_transfer : 0.0011702822521328926
DLL 2024-01-09 16:47:17.762411 - 5 storage : 0.0016443850472569466
DLL 2024-01-09 16:47:17.762447 - 5 tacotron2_items_per_sec : 407.70952041508855
DLL 2024-01-09 16:47:17.762482 - 5 waveglow_items_per_sec : 29821.933051832315
DLL 2024-01-09 16:47:17.762517 - 5 num_mels_per_audio : 750
DLL 2024-01-09 16:47:17.762551 - 5 throughput : 23194.536083814222
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:47:26.320822 - 6 pre_processing : 0.0032453294843435287
DLL 2024-01-09 16:47:26.320936 - 6 tacotron2_latency : 1.9504376584663987
DLL 2024-01-09 16:47:26.320984 - 6 waveglow_latency : 6.601706655696034
DLL 2024-01-09 16:47:26.321025 - 6 latency : 8.552208034321666
DLL 2024-01-09 16:47:26.321064 - 6 type_conversion : 1.7980113625526428e-05
DLL 2024-01-09 16:47:26.321103 - 6 data_transfer : 0.0011230027303099632
DLL 2024-01-09 16:47:26.321141 - 6 storage : 0.0014587454497814178
DLL 2024-01-09 16:47:26.321177 - 6 tacotron2_items_per_sec : 386.5799025808697
DLL 2024-01-09 16:47:26.321213 - 6 waveglow_items_per_sec : 29238.499992037134
DLL 2024-01-09 16:47:26.321248 - 6 num_mels_per_audio : 754
DLL 2024-01-09 16:47:26.321333 - 6 throughput : 22570.077718567805
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:47:34.770021 - 7 pre_processing : 0.0031422488391399384
DLL 2024-01-09 16:47:34.770315 - 7 tacotron2_latency : 1.8702592458575964
DLL 2024-01-09 16:47:34.770371 - 7 waveglow_latency : 6.572657841257751
DLL 2024-01-09 16:47:34.770414 - 7 latency : 8.442971337586641
```

```
DLL 2024-01-09 16:47:34.770455 - 7 type_conversion : 3.156997263431549e-05
DLL 2024-01-09 16:47:34.770498 - 7 data_transfer : 0.0008807918056845665
DLL 2024-01-09 16:47:34.770537 - 7 storage : 0.0014896253123879433
DLL 2024-01-09 16:47:34.770574 - 7 tacotron2_items_per_sec : 401.54860972529684
DLL 2024-01-09 16:47:34.770628 - 7 waveglow_items_per_sec : 29250.87607530315
DLL 2024-01-09 16:47:34.770664 - 7 num_mels_per_audio : 751
DLL 2024-01-09 16:47:34.770699 - 7 throughput : 22771.130247015015
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:47:43.285135 - 8 pre_processing : 0.003208909183740616
DLL 2024-01-09 16:47:43.285248 - 8 tacotron2_latency : 1.893134642392397
DLL 2024-01-09 16:47:43.285296 - 8 waveglow_latency : 6.614848304539919
DLL 2024-01-09 16:47:43.285338 - 8 latency : 8.508040095679462
DLL 2024-01-09 16:47:43.285377 - 8 type_conversion : 1.952052116394043e-05
DLL 2024-01-09 16:47:43.285420 - 8 data_transfer : 0.0015495643019676208
DLL 2024-01-09 16:47:43.285458 - 8 storage : 0.0014447439461946487
DLL 2024-01-09 16:47:43.285495 - 8 tacotron2_items_per_sec : 402.50702878536066
DLL 2024-01-09 16:47:43.285532 - 8 waveglow_items_per_sec : 29490.018669984875
DLL 2024-01-09 16:47:43.285568 - 8 num_mels_per_audio : 762
DLL 2024-01-09 16:47:43.285616 - 8 throughput : 22927.959648316788
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:47:52.092795 - 9 pre_processing : 0.003228388726711273
DLL 2024-01-09 16:47:52.093081 - 9 tacotron2_latency : 1.9571860386058688
DLL 2024-01-09 16:47:52.093135 - 9 waveglow_latency : 6.842815695330501
DLL 2024-01-09 16:47:52.093178 - 9 latency : 8.800061135552824
DLL 2024-01-09 16:47:52.093221 - 9 type_conversion : 1.9189901649951935e-05
DLL 2024-01-09 16:47:52.093264 - 9 data_transfer : 0.0014880048111081123
DLL 2024-01-09 16:47:52.093301 - 9 storage : 0.002112136222422123
DLL 2024-01-09 16:47:52.093337 - 9 tacotron2_items_per_sec : 400.5751035085322
DLL 2024-01-09 16:47:52.093373 - 9 waveglow_items_per_sec : 29330.61607036403
DLL 2024-01-09 16:47:52.093413 - 9 num_mels_per_audio : 784
DLL 2024-01-09 16:47:52.093446 - 9 throughput : 22807.11428118865
Encoder module PASS!
Decoder module PASS!
Tacotron2 module PASS
DLL 2024-01-09 16:48:00.434247 - 10 pre_processing : 0.0034456392750144005
DLL 2024-01-09 16:48:00.434369 - 10 tacotron2_latency : 1.8574912082403898
DLL 2024-01-09 16:48:00.434418 - 10 waveglow_latency : 6.477032033726573
DLL 2024-01-09 16:48:00.434469 - 10 latency : 8.334580512717366
DLL 2024-01-09 16:48:00.434509 - 10 type_conversion : 2.0420178771018982e-05
DLL 2024-01-09 16:48:00.434549 - 10 data_transfer : 0.0010374635457992554
```

```
DLL 2024-01-09 16:48:00.434596 - 10 storage : 0.001502334140241146
DLL 2024-01-09 16:48:00.434635 - 10 tacotron2_items_per_sec : 403.77041714801896
DLL 2024-01-09 16:48:00.434671 - 10 waveglow_items_per_sec : 29643.20679598869
DLL 2024-01-09 16:48:00.434706 - 10 num_mels_per_audio : 750
DLL 2024-01-09 16:48:00.434739 - 10 throughput : 23036.55231442491
Save audio file Pass!
8.439196593035012 22919.50794614992 0.0032260132022202013 0.0030779730528593062 755.5
Throughput average (samples/sec) = 22951.3326
Preprocessing average (seconds) = 0.0032
Postprocessing average (seconds) = 0.0031
Number of mels per audio average = 754.1818181818181
Latency average (seconds) = 8.4131
Latency std (seconds) = 0.2390
Latency cl 50 (seconds) = 8.2778
Latency cl 90 (seconds) = 8.5522
Latency cl 95 (seconds) = 8.8001
Latency cl 99 (seconds) = 8.8001
Latency cl 100 (seconds) = 8.8726
```

本地运行后得到保存的音频文件：

声音清晰且接近人声，但背景音仍有些尖锐的杂音，可能是没有进行移除 waveglow 的 bias 造成的。

5. 实验问题回答

5.1 代码填空

实验中首先要求对空缺代码进行填空，分别实现 model.py 中 tacotron2 的模块和 test_infer.py 中实现 Tacotron2 推断模块。

5.1.1 实现 tacotron2 中的 encoder 模块、Decoder 模块、Tacotron2 模块

1. LocationLayer 类中的 TODOs

对输入进行处理，进行 self.location_conv 操作，得到 F*α：

这一步涉及对输入的注意力权重 (attention_weights_cat) 应用一个卷积层 (self.location_conv)。这是在计算注意力机制中的位置相关特征时的第一步，结果是一个新的特征表示，表示为 F*α。

进行矩阵转置，将原本的行列关系颠倒，得到 (F*α)^T：

该步骤将上一步的输出进行矩阵转置，改变其行和列的关系。这是为了将数据格式化为后续层所需的格式。

对转置的结果进行 self.location_dense 操作，得到 U(F*α)^T：

在这里，转置后的数据被送入一个线性层 (self.location_dense)，进一步转换为所需的特征表示。这是注意力机制中计算位置信息的最后一步。

```
class LocationLayer(nn.Module):
    def __init__(self, attention_n_filters, attention_kernel_size,
                 attention_dim):
        super(LocationLayer, self).__init__()
        padding = int((attention_kernel_size - 1) / 2)
        self.location_conv = ConvNorm(2, attention_n_filters,
                                      kernel_size=attention_kernel_size,
                                      padding=padding, bias=False, stride=1,
                                      dilation=1)
        self.location_dense = LinearNorm(attention_n_filters, attention_dim,
                                         bias=False, w_init_gain='tanh')

    def forward(self, attention_weights_cat):
        # TODO: 对输入进行处理，进行 self.location_conv 操作，得到 F*α
        processed_attention = self.location_conv(attention_weights_cat)
        # TODO: 进行矩阵转置，将原本的行列关系颠倒，得到(F*α)^T
        processed_attention = processed_attention.transpose(1, 2)
        # TODO: 对转置的结果进行 self.location_dense 操作，得到 U(F*α)^T
        processed_attention = self.location_dense(processed_attention)
        return processed_attention
```

2. Attention 类中的 TODOs

对 query 在维度 1 上增加一个新的维度，并使用 self.query_layer 对其进行线性处理，得到 Ws：

这里将解码器的输出 (query) 在一个新的维度上增加一个单位，然后通过一个线性层 (self.query_layer) 处理。这是计算注意力权重时的一部分，目的是将解码器的输出转换成适

合与编码器输出相结合的形式。
使用 self.location_layer 对 attention_weights_cat 进行处理，得到 U(F*α)^T：
这一步将累积的注意力权重 (attention_weights_cat) 通过 LocationLayer 处理，得到一个位置相关的特征表示。
按照公式计算能量函数：
这里是注意力机制的核心，通过组合处理过的查询 (processed_query)、位置特征 (processed_attention_weights) 和编码器的输出 (processed_memory) 来计算能量值。这些能量值决定了解码器在生成下一个输出时应该关注输入序列的哪一部分。
调用 get_alignment_energies 函数得到 alignment：
这一步调用 get_alignment_energies 函数来计算对齐（或注意力权重），这决定了解码器在生成输出时应该更多地关注输入序列的哪些部分。
调用 mask 对 alignment 后面时刻的序列信息进行掩膜操作：
如果提供了掩码 (mask)，这一步将使用它来阻止模型关注序列中的填充部分。这通常在处理不等长序列时非常重要。

```
class Attention(nn.Module):
    def __init__(self, attention_rnn_dim, embedding_dim,
                 attention_dim, attention_location_n_filters,
                 attention_location_kernel_size):
        super(Attention, self).__init__()
        self.query_layer = LinearNorm(attention_rnn_dim, attention_dim,
                                      bias=False, w_init_gain='tanh')
        self.memory_layer = LinearNorm(embedding_dim, attention_dim, bias=False,
                                       w_init_gain='tanh')
        self.v = LinearNorm(attention_dim, 1, bias=False)
        self.location_layer = LocationLayer(attention_location_n_filters,
                                            attention_location_kernel_size,
                                            attention_dim)
        self.score_mask_value = -float("inf")

    def get_alignment_energies(self, query, processed_memory,
                               attention_weights_cat):
        """
        PARAMS
        ------
        query: decoder output (batch, n_mel_channels * n_frames_per_step)
        processed_memory: processed encoder outputs (B, T_in, attention_dim)
        attention_weights_cat: cumulative and prev. att weights (B, 2, max_time)

        RETURNS
        -------
        alignment (batch, max_time)
        """
        # TODO: 对 query 在维度 1 上增加一个新的维度，并使用 self.query_layer 对其进行线性处理，得到 Ws
        processed_query = self.query_layer(query.unsqueeze(1))
```

```
        # TODO: 使用 self.location_layer 对 attention_weights_cat 进行处理，得到 U(F*α)^T
        processed_attention_weights = self.location_layer(attention_weights_cat)
        # TODO: 按照公式计算能量函数（其中，processed_memory 已经在其他函数中完成了全连接的计算 Vh，这里直接作为输入即可）
        energies = self.v(torch.tanh(processed_query + processed_attention_weights + processed_memory))

        energies = energies.squeeze(2)
        return energies

    def forward(self, attention_hidden_state, memory, processed_memory,
                attention_weights_cat, mask):
        """
        PARAMS
        ------
        attention_hidden_state: attention rnn last output
        memory: encoder outputs
        processed_memory: processed encoder outputs
        attention_weights_cat: previous and cummulative attention weights
        mask: binary mask for padded data
        """
        # TODO: 调用 get_alignment_energies 函数得到 alignment
        alignment = self.get_alignment_energies(attention_hidden_state, processed_memory, attention_weights_cat)
          # TODO: 调用 mask 对 alignment 后面时刻的序列信息进行掩膜操作，将填充位置的能量值设置为 score_mask_value
        # alignment = _____________________________________________
        if mask is not None:
            alignment.data.masked_fill_(mask, self.score_mask_value)

        attention_weights = F.softmax(alignment, dim=1)
        attention_context = torch.bmm(attention_weights.unsqueeze(1), memory)
        attention_context = attention_context.squeeze(1)

        return attention_context, attention_weights
```

3. Encoder 类中的 TODOs

对每个卷积层进行循环处理，使用 dropout 和 ReLU 的激活函数：

在这一步，输入数据通过一系列卷积层，每个卷积层后应用 dropout 和 ReLU 激活函数。这是编码器的一部分，用于从输入文本中提取特征。

转置张量 x 的维度：

为了适配 LSTM 的输入格式，需要对经过卷积层处理的数据进行维度转置。

对输入使用 `nn.utils.rnn.pack_padded_sequence` 进行可变长度序列打包，以便 LSTM 处理**：

这一步涉及使用 PyTorch 的 pack_padded_sequence 函数处理数据，以便 LSTM 能够有效

处理不同长度的序列。
运行 LSTM 层：
在这里，打包后的序列数据被送入 LSTM 层进行处理。
对 LSTM 的输出使用 nn.utils.rnn.pad_packed_sequence 进行解包：
LSTM 处理后的输出需要被解包回原来的序列格式，这是通过 pad_packed_sequence 函数实现的。

```
class Encoder(nn.Module):
    """Encoder module:
        - Three 1-d convolution banks
        - Bidirectional LSTM
    """

    def __init__(self, encoder_n_convolutions,
                 encoder_embedding_dim, encoder_kernel_size):
        super(Encoder, self).__init__()

        convolutions = []
        for _ in range(encoder_n_convolutions):
            conv_layer = nn.Sequential(
                ConvNorm(encoder_embedding_dim,
                         encoder_embedding_dim,
                         kernel_size=encoder_kernel_size, stride=1,
                         padding=int((encoder_kernel_size - 1) / 2),
                         dilation=1, w_init_gain='relu'),
                nn.BatchNorm1d(encoder_embedding_dim))
            convolutions.append(conv_layer)
        self.convolutions = nn.ModuleList(convolutions)

        self.lstm = nn.LSTM(encoder_embedding_dim,
                            int(encoder_embedding_dim / 2), 1,
                            batch_first=True, bidirectional=True)

    @torch.jit.export
    def infer(self, x, input_lengths):
        device = x.device
        # TODO: 对每个卷积层进行循环处理，使用 dropout 和 ReLU 的激活函数
        for conv in self.convolutions:
            x = F.dropout(F.relu(conv(x.to(device))), 0.5, self.training)

        # TODO:转置张量 x 的维度，以便与 LSTM 的输入格式匹配；
        x = x = x.transpose(1, 2)

        input_lengths = input_lengths.cpu()
        # TODO: 对输入使用 nn.utils.rnn.pack_padded_sequence 进行可变长度序列打包，以便 LSTM 处理
```

```
        x = nn.utils.rnn.pack_padded_sequence(x, input_lengths, batch_first=True)

        # TODO: 运行 LSTM 层
```

4. Decoder 类中的 TODOs

生成一个全零的 go_frame，作为解码的起始输入：

在解码开始时，需要创建一个全零的帧 (go_frame) 作为初始输入。

初始化解码过程中的各个状态：

在开始解码前，需要初始化各种状态，如注意力相关的隐藏状态、累积的注意力权重等。

使用 prenet 网络对 decoder_input 进行预处理：

这一步使用预网络 (prenet) 对解码器的输入进行处理，这是生成语音输出的第一步。

调用 decode 函数进行一步解码：

这里执行一步解码操作，这包括更新注意力权重、生成下一个声音帧等。

整理解码器的输出：

解码结束后，需要整理输出格式，如调整梅尔频谱的维度等。

```
    @torch.jit.export
        def infer(self, memory, memory_lengths):
            """ Decoder inference
            PARAMS
            ------
            memory: Encoder outputs

            RETURNS
            -------
            mel_outputs: mel outputs from the decoder
            gate_outputs: gate outputs from the decoder
            alignments: sequence of attention weights from the decoder
            """

            # TODO: 生成一个全零的 go_frame，作为解码的起始输入
            decoder_input = torch.zeros(memory.size(0), self.n_mel_channels * self.n_frames_per_step, device=memory.device)

            # TODO: 初始化解码过程中的各个状态
            mask = get_mask_from_lengths(memory_lengths)
            (attention_hidden,
             attention_cell,
             decoder_hidden,
             decoder_cell,
             attention_weights,
             attention_weights_cum,
             attention_context,
             processed_memory) = self.initialize_decoder_states(memory)
            mel_lengths = torch.zeros([memory.size(0)], dtype=torch.int32, device=memory.device)
            not_finished = torch.ones([memory.size(0)], dtype=torch.int32, device=memory.device)
```

```
30.
31.          mel_outputs, gate_outputs, alignments = (
32.              torch.zeros(1), torch.zeros(1), torch.zeros(1))
33.          first_iter = True
34.          while True:
35.              # TODO:使用 prenet 网络对 decoder_input 进行预处理
36.              decoder_input = self.prenet(decoder_input)
37.              # TODO:  调用 decode 函数进行一步解码
38.              (mel_output,
39.               gate_output,
40.               attention_hidden,
41.               attention_cell,
42.               decoder_hidden,
43.               decoder_cell,
44.               attention_weights,
45.               attention_weights_cum,
46.               attention_context) = self.decode(decoder_input, attention_hidden, attention_cell, dec
    oder_hidden,
47.                                                decoder_cell, attention_weights, attention_weights_c
    um, attention_context,
48.                                                memory, processed_memory, mask)
49.              if first_iter:
50.                  mel_outputs = mel_output.unsqueeze(0)
51.                  gate_outputs = gate_output
52.                  alignments = attention_weights
53.                  first_iter = False
54.              else:
55.                  mel_outputs = torch.cat(
56.                      (mel_outputs, mel_output.unsqueeze(0)), dim=0)
57.                  gate_outputs = torch.cat((gate_outputs, gate_output), dim=0)
58.                  alignments = torch.cat((alignments, attention_weights), dim=0)
59.
60.              dec = torch.le(torch.sigmoid(gate_output),
61.                             self.gate_threshold).to(torch.int32).squeeze(1)
62.
63.              not_finished = not_finished * dec
64.              mel_lengths += not_finished
65.
66.              if self.early_stopping and torch.sum(not_finished) == 0:
67.                  break
68.              if len(mel_outputs) == self.max_decoder_steps:
69.                  print("Warning! Reached max decoder steps")
70.                  break
71.
```

```
            decoder_input = mel_output
        # TODO: 整理解码器的输出
        mel_outputs, gate_outputs, alignments = self.parse_decoder_outputs(mel_outputs, gate_outputs, alignments)
        print("Decoder module PASS!")
        return mel_outputs, gate_outputs, alignments, mel_lengths
```

5. Tacotron2 类中的 TODOs

将输入序列 inputs 通过嵌入层进行嵌入：

这里将文本输入转换为嵌入表示，这是处理文本数据的标准步骤。

利用编码器的 infer 函数来对 embedded_inputs 进行编码：

在这一步，嵌入后的文本数据被送入编码器进行进一步处理。

将梅尔频谱输出通过后处理网络（Postnet）进行后处理：

解码器输出的梅尔频谱先通过后处理网络进行质量提升。

将梅尔频谱输出与后处理输出相加，得到最终的梅尔频谱输出：

最后，将解码器的输出与后处理网络的输出相加，以获得最终的梅尔频谱输出。

```
def infer(self, inputs, input_lengths):
        # TODO: 将输入序列 inputs 通过嵌入层进行嵌入，将其进行行列转置，得到 embedded_inputs
        embedded_inputs = self.embedding(inputs).transpose(1, 2)
        # TODO: 利用编码器的 infer 函数来对 embedded_inputs 进行编码，并返回 encoder_outputs
        encoder_outputs = self.encoder.infer(embedded_inputs, input_lengths)
        mel_outputs, gate_outputs, alignments, mel_lengths = self.decoder.infer(
            encoder_outputs, input_lengths)
        # TODO: 将梅尔频谱输出通过后处理网络（Postnet）进行后处理
        mel_outputs_postnet = self.postnet(mel_outputs)
        # TODO: 将梅尔频谱输出与后处理输出相加，得到最终的梅尔频谱输出 mel_outputs_postnet
        mel_outputs_postnet = mel_outputs + mel_outputs_postnet

        BS = mel_outputs_postnet.size(0)
        alignments = alignments.unfold(1, BS, BS).transpose(0, 2)
        print("Tacotron2 module PASS")
        return mel_outputs_postnet, mel_lengths, alignments
```

5.1.2 Tacotron2 的推断模块

1.设定模型推理使用的设备参数为寒武纪 DLP：

这个 TODO 注释指的是需要设置模型推理（inference）时使用的硬件设备。在这个例子中，代码的作者计划使用寒武纪的深度学习处理器（DLP）作为推理设备。

```
# TODO: 设定模型推理使用的设备参数为寒武纪 DLP
    args.device_param = 'mlu'
    args.input_length = 167
```

2. 禁用梯度计算，进行推理过程：

这个 TODO 表示在推理过程中应该禁用 PyTorch 的自动梯度计算功能。这通常在模型推理时做，因为反向传播（计算梯度）是训练过程的一部分，而不是推理过程。禁用梯度计算可

以减少计算量和内存使用，提高推理效率。

```
    # TODO: 禁用梯度计算，进行推理过程。
        with torch.no_grad():
            with MeasureTime(measurements, "latency", args.device_param):
                with MeasureTime(measurements, "tacotron2_latency", args.device_param):
                    # mel, mel_lengths, _ = tacotron2(sequences_padded, input_lengths)
                    mel, mel_lengths, alignment = tacotron2(sequences_padded, input_lengths)

                with MeasureTime(measurements, "waveglow_latency", args.device_param):
                    audios = waveglow.infer(mel, sigma=args.sigma_infer)
                    audios = audios.float()
                    audios = denoiser(audios, strength=args.denoising_strength).squeeze(1)
```

5. 2 思考问题

(1) 本实验使用的都是 32 位浮点数据 (p32) 进行计算，暂未使用混合精度模型，倘若使用混合精度模型推理，会对推理速度有什么影响?

在 Tacotron 2 这样的文本到语音（TTS）项目中，使用 32 位浮点数据（FP32）进行计算意味着所有的数值计算都是在较高的精度下进行。混合精度模型通常是指在模型训练或推理中同时使用 32 位浮点（FP32）和 16 位浮点（FP16）数据进行计算。
使用混合精度进行模型推理可能会对推理速度产生以下影响：

提高性能：16 位浮点运算比 32 位浮点运算更快，因为它们需要的内存带宽更低，而且在支持它们的硬件上，可以更快地完成计算。因此，将部分计算转换为 16 位可以提高整体推理速度。

减少内存使用：FP16 运算占用的内存只有 FP32 的一半。这意味着在内存有限的设备上，可以处理更大的模型或批次大小，进一步提高性能。

硬件依赖性：混合精度推理的效果很大程度上取决于硬件的支持。如果使用的硬件（如 GPU）专门优化了对 FP16 运算的支持，那么使用混合精度可以显著提升推理速度。但如果硬件对 FP16 支持不足，效果可能不明显。

精度的权衡：虽然 FP16 提供了速度和内存使用上的优势，但它的数值表示范围和精度低于 FP32。在某些情况下，这可能导致精度损失，尤其是在涉及较小或较大数值的计算中。然而，在许多实际应用中，这种精度损失是可以接受的，特别是在 TTS 这类任务中。

调整和优化的需要：切换到混合精度可能需要对模型进行某些调整，例如缩放损失函数以适应 FP16 的数值范围，以及确保关键部分的计算仍然在 FP32 下进行以避免精度问题。

总结来说，使用混合精度模型推理在支持该特性的硬件上，通常能够提高推理速度并降低内存消耗，但可能需要针对精度和稳定性做一些优化。在 Tacotron 2 这样的 TTS 系统中，这种方法通常可以在保持合理语音质量的前提下，提高推理效率。

(2) 传统注意力机制都是通过乘法来实现，为什么本实验中的注意力模块使用的加法也能衡量 query 和 key 的相似性?分析两种形式注意力机制的优劣和适用之处。思考为何 tacotron2 中会选择加性注意力?

在神经网络中，注意力机制通常用于确定不同部分的重要性，并据此聚焦于最重要的部分。注意力机制主要有两种形式：乘法（点积）注意力和加法（加性）注意力。它们在衡量

查询（query）和键（key）之间的相似性时采用不同的方法。

乘法（点积）注意力

乘法注意力通过计算查询和键的点积来衡量它们之间的相似性。这种方法直观、简单且计算效率较高，因为它可以利用矩阵乘法优化。但是，点积注意力在处理维度较大的查询和键时可能会出现问题，因为点积随维度的增加而快速增大，这可能导致梯度消失或爆炸问题。

加法（加性）注意力

加法注意力通过先将查询和键连接或组合起来，然后通过一个或多个隐藏层，最后应用一个输出层来计算它们的相似性。这种方法提供了额外的模型灵活性，因为它允许通过隐藏层来学习如何最好地结合查询和键。但是，与点积注意力相比，加性注意力的计算成本较高，因为它涉及更多的参数和复杂的网络结构。

优劣分析

乘法注意力 的优势在于其计算速度快，实现简单。但它在处理高维数据时可能遇到数值稳定性的问题。

加法注意力 提供了更高的灵活性和容错能力，适用于需要更复杂交互的场景，但计算成本和参数量更高。

Tacotron 2 中选择加性注意力的原因

1. 模型复杂性：文本到语音转换是一个复杂的过程，可能需要注意力机制提供更复杂和细粒度的控制。加性注意力通过隐藏层提供了额外的灵活性来学习。这种复杂的关系，这对于处理自然语言和语音合成中的细微变化是有益的。
2. 数值稳定性：在语音合成中，处理的序列通常较长，这可能导致乘法注意力在点积计算中出现数值稳定性问题。加性注意力通过分布在不同层的权重来分散这种风险，提供了更好的稳定性。
3. 模型的泛化能力：加性注意力由于其额外的参数和层，可能在学习如何有效地结合查询和键方面提供更好的泛化能力。这对于处理多样化的文本和语音数据尤其重要。
4. 适应性：加性注意力可以更灵活地适应不同长度的输入和输出序列，这在 TTS 系统中非常重要，因为输入文本和输出语音之间的长度关系并不总是固定的。

6. 实验心得

6.1 深入研究

在进行了本实验的学习后，我还对语音生成领域进行了进一步的探索。由于依照实验架构生成的语音效果最终不甚理想，我搜集资料，尝试使用 HiFi-GAN 替代 WaveGlow 进行声码器的作用，我发现 HiFi-GAN 通过其创新的生成对抗网络（GAN）结构，能够生成更加清晰、细腻的语音。在我的实验中，将 HiFi-GAN 应用于基础架构，显著提高了语音的质量，尤其是在细节和自然度方面。

7. 课程心得