华南理工大学

South China University of Technology

The Experiment Report of *Machine Learning*

SCHOOL: SCHOOL OF SOFTWARE ENGINEERING

SUBJECT: SOFTWARE ENGINEERING

Author:
Xueyu Zhou
Shiyao Sun
Haoxuan Li

Supervisor:
Mingkui Tan

Student ID:
201930384264
201930382376
201930381263

Grade:
2019 Software Engineering Class 1 and 2

December 5, 2021

Speech Synthesis Based on Neural Network

***Abstract*—The objectives of this experiment are to master the basic knowledge of speech signal processing; Understand the application of sequence modeling method in the field of speech synthesis; And understand the basic process of tacotron2 speech synthesis model and practice it.**

I. Introduction

THIS experimental report will be divided into the following four parts. In the first part, Introduce, we will describe the general content of the experimental report. In the second part, Methods and Theory, we will focus on the principle of the model Tacotron2 used in this experiment. In the third part, Experiments we will explain the data set used in the experiment, and focus on the connection between the code used in the experiment and the principle of the second part and the experimental results. In the fourth part, Conclution, we will summarize the difficulties and gains of running the code in this experiment.

II. Methods and Theory

The sound spectrum prediction network used in this experiment is called tacotron2.

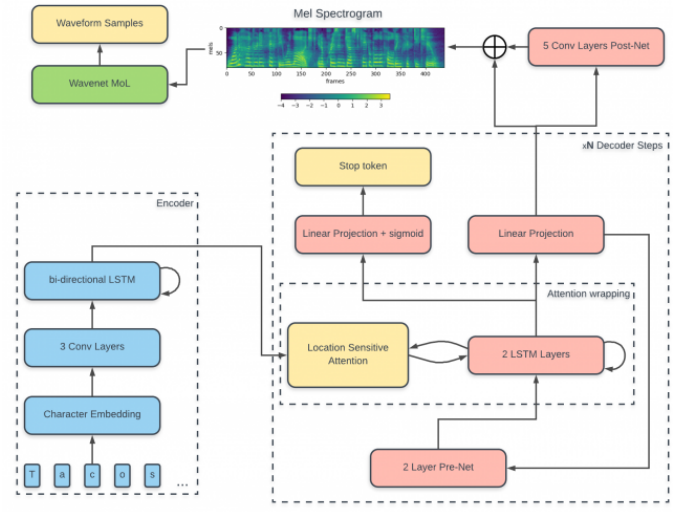

Fig. 1. The Structure of Tacotron2

We can find in the above figure that the network is composed of an encoder and a decoder with attention. And then it is followed by a Wavenet vocoder.

A. Encoder-Decoder

In the original Encoder-Decoder structure, the encoder inputs a sequence or sentence, and then compresses it into a fixed length vector; The decoder uses a fixed length vector to decompress it into a sequence.Just like the following figure 2.

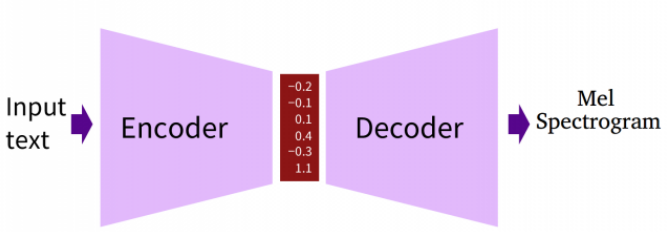

Fig. 2. Encoder-Decoder Model

The most common way is to use RNN to implement encoder and decoder, which is also adopted in Tacotron2.

B. Bidirectional RNN

Bidirectional RNN ensures that the model can perceive forward and backward information at the same time. The bidirectional RNN consists of two independent RNNs. One forward RNN reads the sequence from front to back (from f_1 to f_{T_x}), and the other backward RNN reads the sequence from back to front (from f_{T_x}) to f_1). The final output is the splicing of the two.

In Tacotron2, the encoder maps the input sequence $X = [x_1, x_2, \ldots, x_{T_x}]$to the sequence $H = [h_1, h_2, \ldots, h_{T_x}]$, where the sequence h is called "encoder hidden state".

Also, each input component X_i is a character. The encoder of Tacotron2 is a module formed by a three-layer convolution layer followed by a bidirectional LSTM layer. In Tacotron2, the convolution layer gives the neural network the ability to perceive the context similar to N-Gram. The main reason for using convolution layer to obtain context here is that RNN is difficult to capture long-term dependence in practice, and the use of convolution layer makes the model more robust to silent characters (such as' k 'in' know ').

The word embedded character sequence is first sent to three convolution layers to extract the context information, and then sent to a bidirectional LSTM to generate the encoder hidden state, namely:

$$f_e = ReLU(F_3 * ReLU(F_2 * ReLU(F_1 * \overline{E}(X))))$$

$$H = EncoderRecurrency(f_e)$$

Among them, F_1, F_2 and F_3 are three convolution cores, $ReLU$ is the nonlinear activation on each convolution layer, $\overline{E}(X)$ which means embedding the character sequence x, and $EncoderRecurrency$ means bidirectional LSTM.

After the implicit state of the encoder is generated, it will be sent to the attention network to generate a context vector.

C. Attention Mechanism

Attention is used as a bridge between encoder and decoder. It is essentially a matrix composed of context weight vectors.

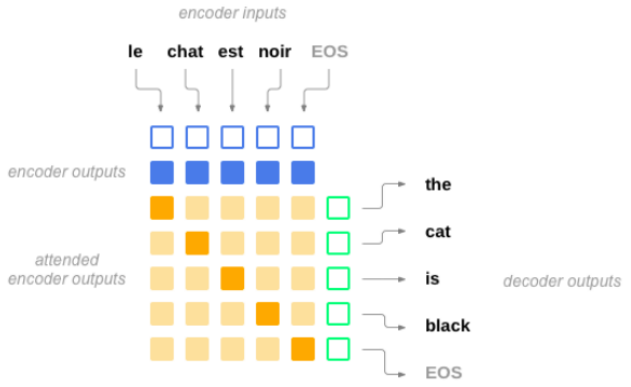

Fig. 3. Attention Matrix

$$Attention(Query, Source) = \sum_{i=1}^{L_x} similarity(Query, Key_i) * Value \tag{1}$$

In machine translation (NMT), the combination of key and value in souce refers to the same thing, that is, the semantic code corresponding to each word in the input sentence.

General calculation steps:

Step 1: Key and Value Similarity Measurement:

dot product:

$$Similarity(Query, Key) = QueryKey$$

cos similarity:

$$Similarity(Query, Key) = \frac{QueryKey_i}{||Query|| * ||Key_i||}$$

MLP network:

$$Similarity(Query, Key_i) = MLP(Query, Key_i)$$

Step 2: Softmax Normalization for Attention Weights

$$a_i = softmax(sim_i) = \frac{e^{sim_i}}{\sum_{j=1}^{L_x} e^{sim_j}}$$

Step 3: Calculate the Attention Value (Context Vector):

$$Attention(Query, Key) = \sum_{i=1}^{L_x} a_i Value_i$$

Also, in Tacotron2, attention computation occurs at each decoder time step, which includes the following stages:

1.The target hidden state (shown in the green box above) is "compared" with each source state (shown in the blue box above) to generate attention weights or alignments:

$$\alpha_{ts} = \frac{exp(score(h_t, \overline{h_s}))}{\sum_{s'=1}^{S} exp(score(h_t, \overline{h_{s'}}))}$$

Where h_t is the target hidden state, $\overline{h_s}$is the source state, and the score function is often called "energy", so it can be expressed as e. Different $score$ functions determine different types of attention mechanisms.

2.Based on the attention weight, the context vector is calculated as the weighted average of the source state:

$$c_t = \sum_s \alpha_{ts} \overline{h_s}$$

3.The attention vector is used as the input of the next time step.

In addition, the attention mechanism used by Tacotron2 is **Location Sensitive Attention**:

$$e_{i,j} = score(s_i, c\alpha_{i-1}, h_j) = v_a^T \, tanh(Ws_i + Vh_j + Uf_{i,j} + b)$$

Where s_i is the current decoder hidden state instead of the previous decoder hidden state, and the offset value b is initialized to 0. The position feature f_i uses the cumulative attention weight $c\alpha_i$ convolution:

$$\begin{aligned} f_i &= F * c\alpha_{i-1} \\ c\alpha_i &= \sum_{j=1}^{i-1} \alpha_j \end{aligned} \tag{2}$$

D. Decoder

The decoding process starts from inputting the output spectrum of the previous step or the real spectrum of the previous step to prenet. The process is as follows:

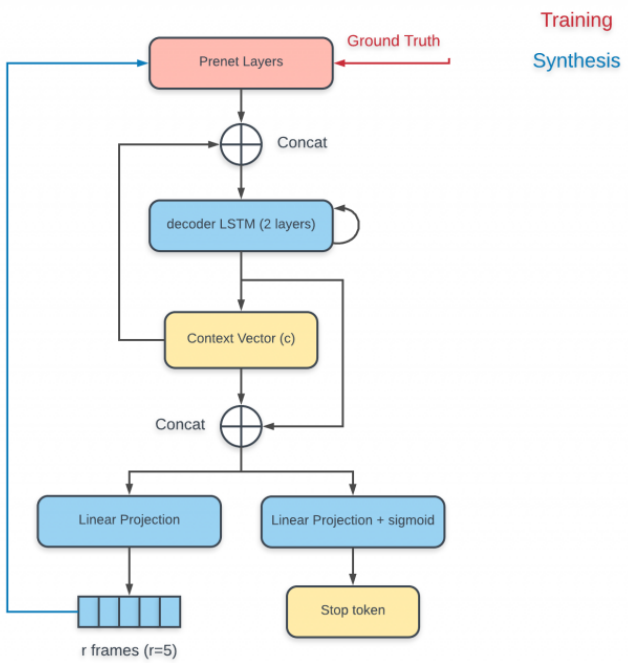

Fig. 4. Structure of Decorder

The output of PreNet is spliced with the context vector calculated by using the output of the previous decoding step, and then the whole is sent to the RNN decoder. The output of the RNN decoder is used to calculate the new context vector. Finally, the newly calculated context vector is spliced with

the output of the decoder and sent to the projection layer to predict the output. There are two forms of output, one is the sound spectrum frame, and the other is the probability of $< stoptoken >$, which is a simple binary classification problem to determine whether the decoding process is over. Using the reduction factor, that is, only r (reduction factor) Mel spectral frames are allowed to be predicted in each decoding step, which can effectively accelerate the calculation and reduce the memory occupation.

E. PostNet

Once the decoder completes decoding, the predicted Mel spectrum is sent to a series of convolution layers to improve the generation quality.

The post-processing network uses residual calculation:

$$y_{final} = y + y_r$$

Where y is the original input.

Where,

$$y_r = PostNet(y) = W_{ps} f_{ps} + b_{ps}$$

Where $f_{ps} = F_{ps,i} * x$, x is the output of the previous convolution layer or decoder, and F is convolution

F. Training

$$loss = \frac{1}{n}\sum_{i=1}^{n}(y_{real,i} - y_i)^2 + \frac{1}{n}\sum_{i=1}^{n}(y_{real,i} - y_{final,i})^2 + \lambda \sum_{j=1}^{p} w_j^2$$

Where, $y_{real,i}$ is the real sound spectrum, y_i, $y_{final,i}$ are the sound spectrum before and after entering the post-processing network respectively, and N is the number of samples in batch, λ Is the regularization parameter, p is the total number of parameters, and w is the parameter in the neural network. Note that you do not need to regularize the offset value.

G. WaveNet Vocoder

Tacontron2 the original paper used a modified WaveNet, which inversely transformed the Mel spectrum feature expression into time-domain waveform samples, but the Waveflow Vocoder is also available now.

III. Experiments

A. Dataset

This experiment uses the English speech synthesis public data set LJSpeech, which contains 13000 short audio, each audio has corresponding text, and the total time is 24 hours. In this experiment, we divide the data set into training set with 12500 audio, test set with 500 audio and validation set with 100 audio.

B. Implementation

As described in the **Methods and Theroy** section above, we will show each part of the code in turn in the next section.

1) Encorder: The input character is encoded into a 512 dimensional character vector.

Then it passes through a three-layer convolution. Each convolution layer contains 512 5x1 convolution cores, that is, each convolution core spans 5 characters. The convolution layer will conduct long-span context modeling for the input character sequence (similar to N-Grams). Here, the convolution layer is used to obtain the context, mainly because it is difficult for RNN to capture long-term dependence in practice; The convolution layer is followed by batch normalization, which is activated by ReLU.

The output of the last convolution layer is transmitted to a bidirectional LSTM layer to generate coding features. The LSTM contains 512 units (256 units in each direction).

```
class Encoder(nn.Module):
    """Encoder module:
        - Three 1-d convolution banks
        - Bidirectional LSTM
    """
    def __init__(self):
        super(Encoder, self).__init__()

        convolutions = []
        for _ in range(config.encoder_n_\
        convolutions):
            conv_layer = nn.Sequential(
                ConvNorm(config.encoder_\
                embedding_dim,
                         config.encoder_\
                         embedding_dim,
                         kernel_size=config\
                         .encoder_kernel\
                         _size, stride=1,
                         padding=int\
                         ((config.encoder\
                         _kernel_size\
                         - 1) / 2),
                         dilation=1\
                         , w_init_gain='relu'),
                nn.BatchNorm1d\
                (config.encoder_\
                embedding_dim))
            convolutions.append\
            (conv_layer)
        self.convolutions = \
        nn.ModuleList(convolutions)

        self.lstm = nn.LSTM\
        (config.encoder_embedding_dim,
                           int(config.\
                           encoder_\
                           embedding_dim / 2), 1,
                           batch_first\
                           =True, bidirectional=
```

2) Attention Mechanism: Tacotron2 uses a location-based attention mechanism, which is an extension of the previous attention mechanism; In this way, the cumulative attention weight of the previous decoding process can be used as an additional feature, so the model remains consistent when moving forward along the input sequence, reducing the potential subsequence repetition or omission in the decoding process. The position features are convoluted by 32 1-dimensional convolution kernels with a length of 31, and then the input sequence and position features are projected to the 128 di-

mensional hidden layer for characterization.

```
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) target hidden state

        processed_memory: processed encoder outputs
        (B, T_in, attention_dim) source state

        attention_weights_cat: cumulative an
        d prev. att weights (B, 2, max_time)
        RETURNS
        -------
        alignment (batch, max_time)
        """

        processed_query = self.query\
        _layer(query.unsqueeze(1))
        processed_attention_weights\
        = self.location_layer(atten\
        tion_weights_cat)
        energies = self.v(torch.tanh\
        (processed_query + processed\
        _attention_weights + processed_memory))

        energies = energies.squeeze(-1)
        return energies
```

3) Location Feature: Calculate the location feature: the location feature is obtained by convolution of the previously accumulated attention weight.

```
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 = Conv\
        Norm(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):
        processed_attention = self\
        .location_conv(attention_weights_cat)
        processed_attention = proc\
        essed_attention.transpose(1, 2)
        processed_attention = self\
        .location_dense(processed_attention)
        return processed_attention
```

4) Main Part of Decorder: The decoder is an autoregressive recurrent neural network, which predicts the output spectrogram from the encoded input sequence, one frame at a time.

The spectrum predicted in the previous step is first introduced into a "PreNet", a double-layer full connection layer composed of 256 hidden ReLU units in each layer. As an information bottleneck layer, PreNet is necessary for learning attention.

The output of PreNet and attention context vector are spliced together and transmitted to a two-layer stacked one-way LSTM composed of 1024 units. The output of LSTM is spliced with the attention context vector again, and then the target spectrum frame is predicted through a linear projection.

Finally, the target spectrum frame is predicted through a 5-layer convolution "post net", and a residual is superimposed on the spectrum frame before convolution to improve the whole process of spectrum reconstruction. Each layer of post net consists of 512 5x1 convolution cores, followed by batch normalization layer. Except for the last layer of convolution, each layer of batch normalization is activated by tanh.

Parallel to the prediction of spectral frames, the output of the decoder LSTM is spliced with the attention context vector, projected into a scalar and passed to the sigmoid activation function to predict the probability of whether the output sequence has been completed.

In the code of the main part of the decoder, you can see which layers the decoder is composed of, PreNet (preprocessing layer), attention_rnn, attention_layer (attention network), decoder_RNN (decoder LSTM), linear_project (linear projection layer), gate_layer (judge whether the prediction is over).

```
class Decoder(nn.Module):
    def __init__(self):
        super(Decoder, self)\
        __init__()
        self.n_mel_channels =\
        config.n_mel_channels
        self.n_frames_per_ste\
        = config.n_frames_per_step
        self.encoder_embedding_\
        dim = config.encoder_embedding_dim
        self.attention_rnn_dim \
        = config.attention_rnn_dim
        self.decoder_rnn_dim = \
        config.decoder_rnn_dim
        self.prenet_dim = config\
        .prenet_dim
        self.max_decoder_steps =\
        config.max_decoder_steps
        self.gate_threshold = \
        config.gate_threshold
        self.p_attention_dropout\
        = config.p_attention_dropout
        self.p_decoder_dropout\
        = config.p_decoder_dropout

        self.prenet = Prenet(config\
```

```
.n_mel_channels * config.n_\
frames_per_step,[config.pre\
net_dim, config.prenet_dim])

self.attention_rnn = nn.LSTMCell(
    config.prenet_dim + \
    config.encoder_embedding_dim,
    config.attention_rnn_dim)

self.attention_layer = Attention(
    config.attention_rnn_\

    dim, config.encoder_\
    embedding_dim,
    config.attention_dim, \
    config.attention_\
    location_n_filters,
    config.attention_\
    location_kernel_size)

self.decoder_rnn = nn.LSTMCell\
(config.attention_rnn_dim + config\
.encoder_embedding_dim,config.\
decoder_rnn_dim, 1)

self.linear_projection = \
LinearNorm(config.decoder_rnn_dim\
+ config.encoder_embedding_dim
,config.n_mel_channels * config.\
n_frames_per_step)

self.gate_layer = LinearNorm(
    config.decoder_rnn_dim\
    + config.encoder_embedding_dim, 1,
    bias=True, w_init_gain='sigmoid')
```

5) PreNet: See the principle above for details, and see the implementation of Prenet class in Tacotron2 / Model.py file for code, which is omitted here.

6) PostNet: See the principle above for details, and see the implementation of Postnet class in Tacotron2 / Model.py file for code, which is omitted here.

C. Result

Due to time reasons, we only trained 5 models(we can find it in the table below) and made 3 groups of simple comparative tests were conducted in this experiment, namely, comparing the impact of encoder kernel size and learning rate on loss, and comparing the loss difference and time difference between multiple GPUs and single GPUs, also, comparing the effect of given model with our training model on validation set.

TABLE I
PARAMETER INITIALIZATION

Number	LearningRate	KernelSize	Epoch	GPUnumber	OtherParams
1	0.005	5	25	1	SAME
2	0.001	5	25	1	SAME
3	0.001	10	25	1	SAME
4	0.001	5	200	1	SAME
5	0.001	5	200	2	SAME

The following table and figures show the effects of learning rate and encoder kernel size on the loss function.

We use the symbol EX_1 to represent the experimental group 1 with a learning rate of 0.005 and a encoder kernel size of 5 and use the symbol EX_2 to represent the experimental group 2 with a learning rate of 0.001 and a encoder kernel size of 5, also, use the symbol EX_3 to represent the experimental group 3 with a learning rate of 0.001 and a encoder kernel size of 10.

So, according to the result below, we can easily find that the learning rate of 0.005 is too big for training the model, so its loss is higher than the learning rate of 0.001; but about the encoder kernel size, there is no enough reason to prove that it will affect the experimental results positively or negatively.

TABLE II
EFFECTS OF LEARNING RATE AND ENCODER KERNEL SIZE ON THE LOSS FUNCTION

E	I	EX_1	EX_2	EX_3
2	127	1.538549	1.290112	1.317796
4	257	1.312044	1.290112	1.073937
7	127	1.156257	0.835678	0.835184
9	257	1.008528	0.698864	0.704667
12	127	1.060405	0.678221	0.677272
14	257	0.956149	0.648457	0.648590
17	127	0.992673	0.666788	0.665517
19	257	1.037558	0.701215	0.697902
22	126	0.859282	0.573909	0.573638
24	259	0.867594	0.566261	0.578215

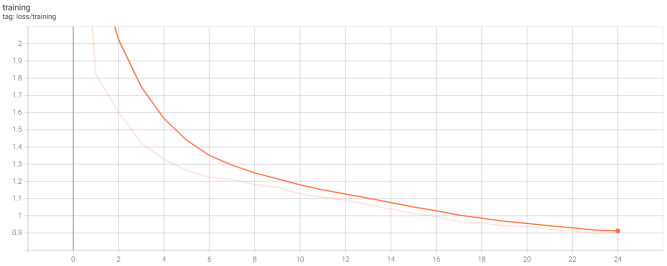

Fig. 5. EX_1's Loss

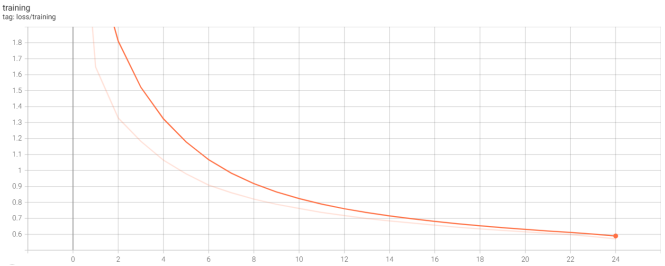

Fig. 6. EX_2's Loss

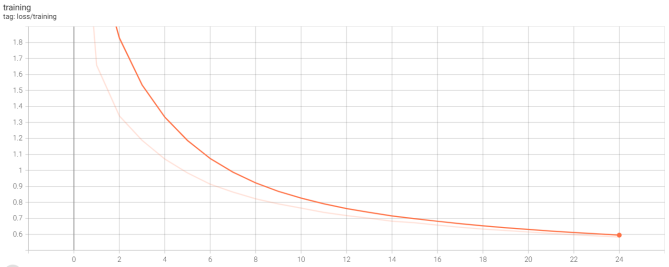

Fig. 7. EX_3's Loss

The following tables show the effects of MultiGPUs and Single GPU on the loss function and implement time.

It can be seen from the experimental results that single and dual GPUs have no effect on the time of each iteration, but each epoch of single GPU corresponds to 260 iterations, while dual GPUs correspond to 130 iterations, which is equivalent to twice the overall running time of dual GPUs.

It can also be seen from the results that the number of GPUs has no effect on the loss size of the same iteration.

TABLE III
THE LOSS FUNCTION AND TIME OF SINGLE GPU

E	I	Loss	Time for One Iteration
0	100	3.191367	3.0997
20	100	0.631788	3.0821
40	100	0.559845	3.1558
60	100	0.474369	3.1621
80	100	0.451346	3.1106
100	100	0.410702	3.2048
120	100	0.362405	3.1228
140	100	0.385817	3.1379
160	100	0.370358	3.2610
180	100	0.329293	3.1553

TABLE IV
THE LOSS FUNCTION AND TIME OF MULTIGPUS(GPU0)

E	I	Loss	Time for One Iteration
0	100	3.083026	4.4806
20	100	0.722199	3.2935
40	100	0.585557	3.2601
60	100	0.522424	3.2648
80	100	0.485112	3.3261
100	100	0.454771	3.5437
120	100	0.414174	3.3083
140	100	0.383420	3.5015
160	100	0.356977	3.2795
180	100	0.344687	3.3266

TABLE V
THE LOSS FUNCTION AND TIME OF MULTIGPUS(GPU1)

E	I	Loss	Time for One Iteration
0	100	3.083026	4.0010
20	100	0.722199	3.3749
40	100	0.585557	3.4206
60	100	0.522424	3.3923
80	100	0.485112	3.3662
100	100	0.454771	3.6853
120	100	0.414174	3.3822
140	100	0.383420	3.6039
160	100	0.356977	3.3600
180	100	0.344687	3.3499

The following are some visualization results on validation of the model we trained comparing with the given model (Of course, we trained 5 models and we choose the best one to compare with the given) .

From the final results, because the time of this experiment is not sufficient, we only ran 200 epochs, so there is still a large gap between the trained model and the given 500 epochs model, especially in the peak value. However, compared with the original waveform, we can still see the training results.

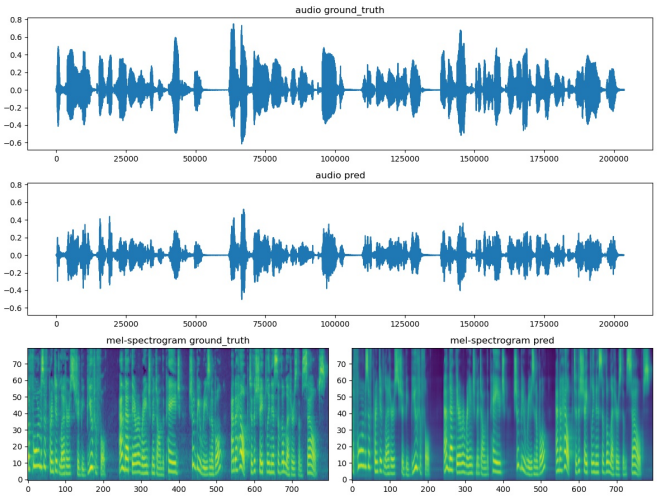

Fig. 8. Audio LJ001-0015 with Given Model

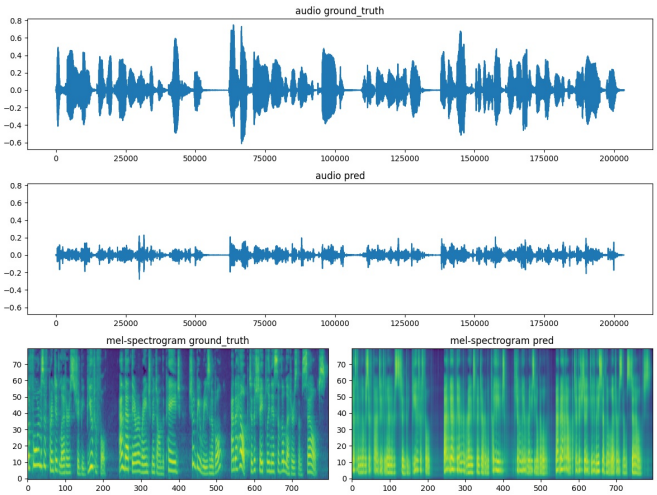

Fig. 9. Audio LJ001-0015 with Our Model

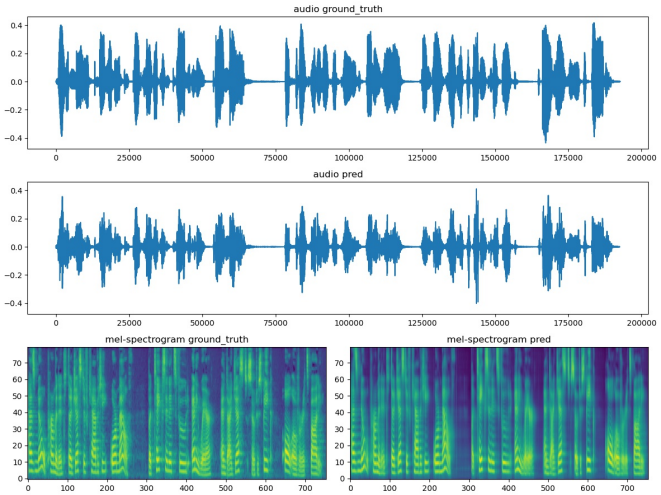

Fig. 10. Audio LJ025-0081 with Given Model

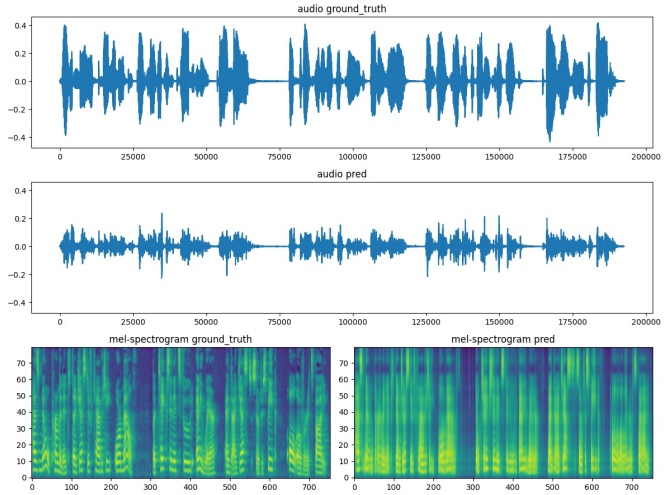

Fig. 11. Audio LJ025-0081 with Our Model

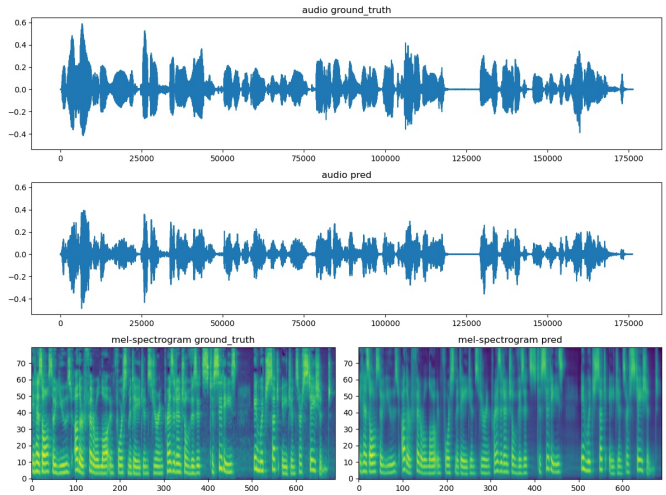

Fig. 12. Audio LJ050-0235 with Given Model

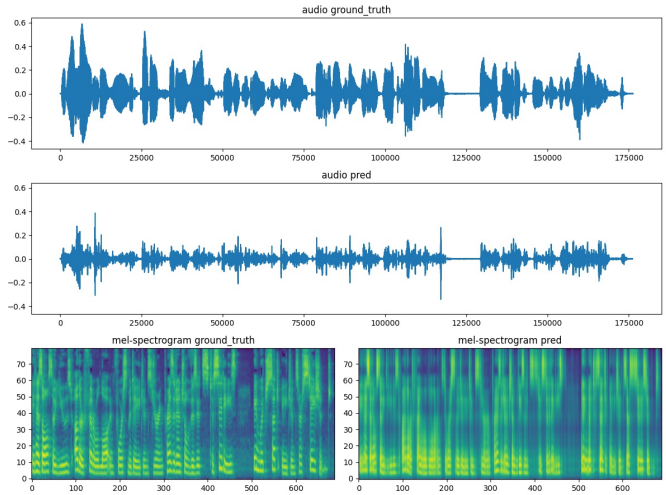

Fig. 13. Audio LJ050-0235 with Our Model

IV. Conclusion

The most difficult part of this experiment is during running the code. Due to the fact that we rented the pytoch version and code version of 3090 graphics card in this experiment, the running code of our team is loss, which has been around 700 and can not be reduced. Later, after a series of efforts, we found that only the first layer w of the model has a certain value, and all the other layers are Nan. Later, change line 36 of the utils.py file from:

```
mask = (ids < lengths.unsqueeze(1)).byte()
```

to:

```
mask = (ids < lengths.unsqueeze(1)).bool()
```

And finally we solve the problem.

When finally verifying the model, we also modified the information.py file from testing only one statement to testing the statements of the whole verification set.

In addition, We have gained a lot from this experiment. This is the first time we rent a GPU server, and we have a better understanding of the commands of the Linux system. At the same time, the content used in Experiment 8 is what we talked about in the second half of the class. In the process of doing the experiment, our team members reviewed a series of contents described in class, including CNN, RNN, attention mechanism, etc. Although the process was full of twists and turns, the experiment was successfully completed.