درس:

پردازش سیگنال‌های دیجیتال

موضوع p24:

Introducing Translatotron: An End-to-End Speech-to-Speech Translation Model

استاد :

جناب آقای دکتر مهدی اسلامی

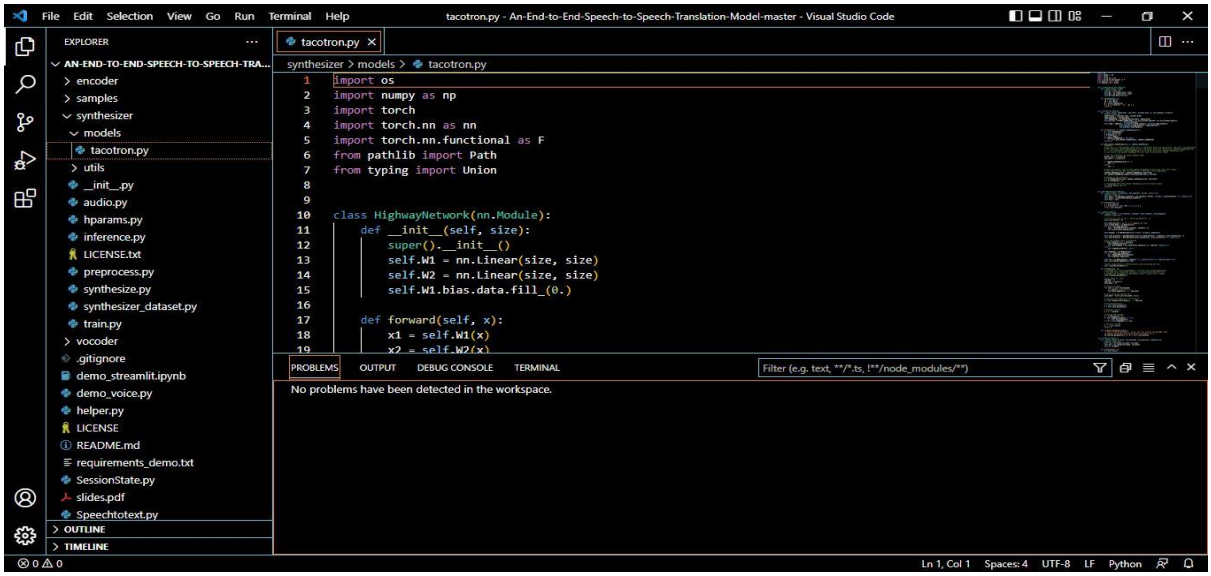
دانشجو:

حمیدرضا پورمحمّد

شماره دانشجویی :

۴۰۰۱۴۱۴۰۱۱۱۰۳۳

پاییز ۱۴۰۱




```python
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from pathlib import Path
from typing import Union


class HighwayNetwork(nn.Module):
    def __init__(self, size):
        super().__init__()
        self.W1 = nn.Linear(size, size)
        self.W2 = nn.Linear(size, size)
        self.W1.bias.data.fill_(0.)

    def forward(self, x):
        x1 = self.W1(x)
        x2 = self.W2(x)
        g = torch.sigmoid(x2)
        y = g * F.relu(x1) + (1. - g) * x
        return y


class Encoder(nn.Module):
    def __init__(self, embed_dims, num_chars, encoder_dims, K, num_highways,
dropout):
        super().__init__()
        prenet_dims = (encoder_dims, encoder_dims)
        cbhg_channels = encoder_dims
        self.embedding = nn.Embedding(num_chars, embed_dims)
```

```
        self.pre_net = PreNet(embed_dims, fc1_dims=prenet_dims[0],
fc2_dims=prenet_dims[1],
                              dropout=dropout)
        self.cbhg = CBHG(K=K, in_channels=cbhg_channels,
channels=cbhg_channels,
                         proj_channels=[cbhg_channels, cbhg_channels],
                         num_highways=num_highways)

    def forward(self, x, speaker_embedding=None):
        x = self.embedding(x)
        x = self.pre_net(x)
        x.transpose_(1, 2)
        x = self.cbhg(x)
        if speaker_embedding is not None:
            x = self.add_speaker_embedding(x, speaker_embedding)
        return x

    def add_speaker_embedding(self, x, speaker_embedding):
        # SV2TTS
        # The input x is the encoder output and is a 3D tensor with size
(batch_size, num_chars, tts_embed_dims)
        # When training, speaker_embedding is also a 2D tensor with size
(batch_size, speaker_embedding_size)
        #     (for inference, speaker_embedding is a 1D tensor with size
(speaker_embedding_size))
        # This concats the speaker embedding for each char in the encoder
output

        # Save the dimensions as human-readable names
        batch_size = x.size()[0]
        num_chars = x.size()[1]

        if speaker_embedding.dim() == 1:
            idx = 0
        else:
            idx = 1

        # Start by making a copy of each speaker embedding to match the input
text length
        # The output of this has size (batch_size, num_chars * tts_embed_dims)
        speaker_embedding_size = speaker_embedding.size()[idx]
        e = speaker_embedding.repeat_interleave(num_chars, dim=idx)

        # Reshape it and transpose
        e = e.reshape(batch_size, speaker_embedding_size, num_chars)
        e = e.transpose(1, 2)

        # Concatenate the tiled speaker embedding with the encoder output
```

```
        x = torch.cat((x, e), 2)
        return x


class BatchNormConv(nn.Module):
    def __init__(self, in_channels, out_channels, kernel, relu=True):
        super().__init__()
        self.conv = nn.Conv1d(in_channels, out_channels, kernel, stride=1,
padding=kernel // 2, bias=False)
        self.bnorm = nn.BatchNorm1d(out_channels)
        self.relu = relu

    def forward(self, x):
        x = self.conv(x)
        x = F.relu(x) if self.relu is True else x
        return self.bnorm(x)


class CBHG(nn.Module):
    def __init__(self, K, in_channels, channels, proj_channels, num_highways):
        super().__init__()

        # List of all rnns to call `flatten_parameters()` on
        self._to_flatten = []

        self.bank_kernels = [i for i in range(1, K + 1)]
        self.conv1d_bank = nn.ModuleList()
        for k in self.bank_kernels:
            conv = BatchNormConv(in_channels, channels, k)
            self.conv1d_bank.append(conv)

        self.maxpool = nn.MaxPool1d(kernel_size=2, stride=1, padding=1)

        self.conv_project1 = BatchNormConv(len(self.bank_kernels) * channels,
proj_channels[0], 3)
        self.conv_project2 = BatchNormConv(proj_channels[0], proj_channels[1],
3, relu=False)

        # Fix the highway input if necessary
        if proj_channels[-1] != channels:
            self.highway_mismatch = True
            self.pre_highway = nn.Linear(proj_channels[-1], channels,
bias=False)
        else:
            self.highway_mismatch = False

        self.highways = nn.ModuleList()
        for i in range(num_highways):
            hn = HighwayNetwork(channels)
```

```
            self.highways.append(hn)

        self.rnn = nn.GRU(channels, channels // 2, batch_first=True,
bidirectional=True)
        self._to_flatten.append(self.rnn)

        # Avoid fragmentation of RNN parameters and associated warning
        self._flatten_parameters()

    def forward(self, x):
        # Although we `_flatten_parameters()` on init, when using DataParallel
        # the model gets replicated, making it no longer guaranteed that the
        # weights are contiguous in GPU memory. Hence, we must call it again
        self._flatten_parameters()

        # Save these for later
        residual = x
        seq_len = x.size(-1)
        conv_bank = []

        # Convolution Bank
        for conv in self.conv1d_bank:
            c = conv(x) # Convolution
            conv_bank.append(c[:, :, :seq_len])

        # Stack along the channel axis
        conv_bank = torch.cat(conv_bank, dim=1)

        # dump the last padding to fit residual
        x = self.maxpool(conv_bank)[:, :, :seq_len]

        # Conv1d projections
        x = self.conv_project1(x)
        x = self.conv_project2(x)

        # Residual Connect
        x = x + residual

        # Through the highways
        x = x.transpose(1, 2)
        if self.highway_mismatch is True:
            x = self.pre_highway(x)
        for h in self.highways: x = h(x)

        # And then the RNN
        x, _ = self.rnn(x)
        return x
```

```
    def _flatten_parameters(self):
        """Calls `flatten_parameters` on all the rnns used by the WaveRNN.
Used
        to improve efficiency and avoid PyTorch yelling at us."""
        [m.flatten_parameters() for m in self._to_flatten]

class PreNet(nn.Module):
    def __init__(self, in_dims, fc1_dims=256, fc2_dims=128, dropout=0.5):
        super().__init__()
        self.fc1 = nn.Linear(in_dims, fc1_dims)
        self.fc2 = nn.Linear(fc1_dims, fc2_dims)
        self.p = dropout

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = F.dropout(x, self.p, training=True)
        x = self.fc2(x)
        x = F.relu(x)
        x = F.dropout(x, self.p, training=True)
        return x


class Attention(nn.Module):
    def __init__(self, attn_dims):
        super().__init__()
        self.W = nn.Linear(attn_dims, attn_dims, bias=False)
        self.v = nn.Linear(attn_dims, 1, bias=False)

    def forward(self, encoder_seq_proj, query, t):

        # print(encoder_seq_proj.shape)
        # Transform the query vector
        query_proj = self.W(query).unsqueeze(1)

        # Compute the scores
        u = self.v(torch.tanh(encoder_seq_proj + query_proj))
        scores = F.softmax(u, dim=1)

        return scores.transpose(1, 2)


class LSA(nn.Module):
    def __init__(self, attn_dim, kernel_size=31, filters=32):
        super().__init__()
        self.conv = nn.Conv1d(1, filters, padding=(kernel_size - 1) // 2,
kernel_size=kernel_size, bias=True)
        self.L = nn.Linear(filters, attn_dim, bias=False)
```

```
        self.W = nn.Linear(attn_dim, attn_dim, bias=True) # Include the
attention bias in this term
        self.v = nn.Linear(attn_dim, 1, bias=False)
        self.cumulative = None
        self.attention = None

    def init_attention(self, encoder_seq_proj):
        device = next(self.parameters()).device  # use same device as
parameters
        b, t, c = encoder_seq_proj.size()
        self.cumulative = torch.zeros(b, t, device=device)
        self.attention = torch.zeros(b, t, device=device)

    def forward(self, encoder_seq_proj, query, t, chars):

        if t == 0: self.init_attention(encoder_seq_proj)

        processed_query = self.W(query).unsqueeze(1)

        location = self.cumulative.unsqueeze(1)
        processed_loc = self.L(self.conv(location).transpose(1, 2))

        u = self.v(torch.tanh(processed_query + encoder_seq_proj +
processed_loc))
        u = u.squeeze(-1)

        # Mask zero padding chars
        u = u * (chars != 0).float()

        # Smooth Attention
        # scores = torch.sigmoid(u) / torch.sigmoid(u).sum(dim=1,
keepdim=True)
        scores = F.softmax(u, dim=1)
        self.attention = scores
        self.cumulative = self.cumulative + self.attention

        return scores.unsqueeze(-1).transpose(1, 2)


class Decoder(nn.Module):
    # Class variable because its value doesn't change between classes
    # yet ought to be scoped by class because its a property of a Decoder
    max_r = 20
    def __init__(self, n_mels, encoder_dims, decoder_dims, lstm_dims,
                 dropout, speaker_embedding_size):
        super().__init__()
        self.register_buffer("r", torch.tensor(1, dtype=torch.int))
        self.n_mels = n_mels
        prenet_dims = (decoder_dims * 2, decoder_dims * 2)
```

```
        self.prenet = PreNet(n_mels, fc1_dims=prenet_dims[0],
fc2_dims=prenet_dims[1],
                             dropout=dropout)
        self.attn_net = LSA(decoder_dims)
        self.attn_rnn = nn.GRUCell(encoder_dims + prenet_dims[1] +
speaker_embedding_size, decoder_dims)
        self.rnn_input = nn.Linear(encoder_dims + decoder_dims +
speaker_embedding_size, lstm_dims)
        self.res_rnn1 = nn.LSTMCell(lstm_dims, lstm_dims)
        self.res_rnn2 = nn.LSTMCell(lstm_dims, lstm_dims)
        self.mel_proj = nn.Linear(lstm_dims, n_mels * self.max_r, bias=False)
        self.stop_proj = nn.Linear(encoder_dims + speaker_embedding_size +
lstm_dims, 1)

    def zoneout(self, prev, current, p=0.1):
        device = next(self.parameters()).device  # Use same device as
parameters
        mask = torch.zeros(prev.size(), device=device).bernoulli_(p)
        return prev * mask + current * (1 - mask)

    def forward(self, encoder_seq, encoder_seq_proj, prenet_in,
                hidden_states, cell_states, context_vec, t, chars):

        # Need this for reshaping mels
        batch_size = encoder_seq.size(0)

        # Unpack the hidden and cell states
        attn_hidden, rnn1_hidden, rnn2_hidden = hidden_states
        rnn1_cell, rnn2_cell = cell_states

        # PreNet for the Attention RNN
        prenet_out = self.prenet(prenet_in)

        # Compute the Attention RNN hidden state
        attn_rnn_in = torch.cat([context_vec, prenet_out], dim=-1)
        attn_hidden = self.attn_rnn(attn_rnn_in.squeeze(1), attn_hidden)

        # Compute the attention scores
        scores = self.attn_net(encoder_seq_proj, attn_hidden, t, chars)

        # Dot product to create the context vector
        context_vec = scores @ encoder_seq
        context_vec = context_vec.squeeze(1)

        # Concat Attention RNN output w. Context Vector & project
        x = torch.cat([context_vec, attn_hidden], dim=1)
        x = self.rnn_input(x)
```

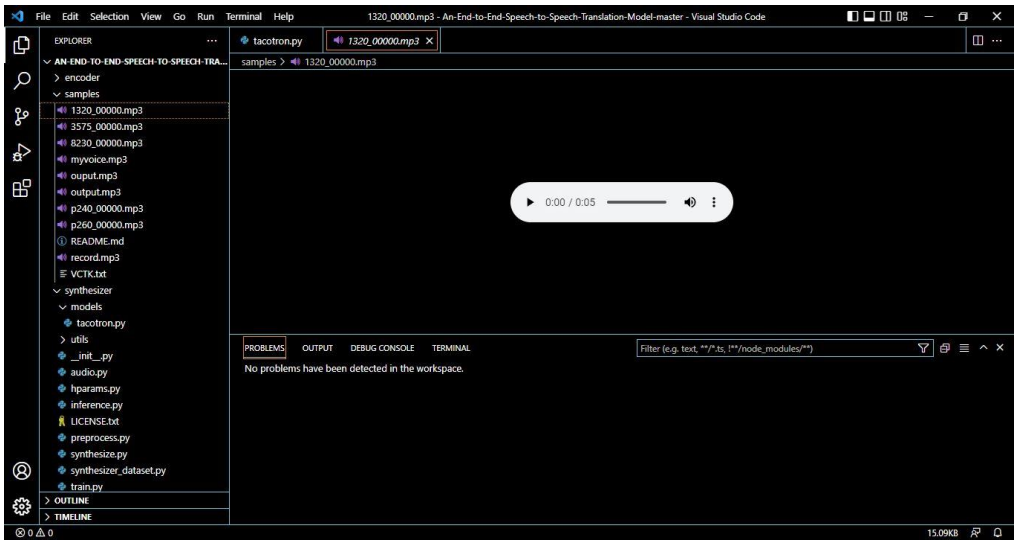
```
        # Compute first Residual RNN
        rnn1_hidden_next, rnn1_cell = self.res_rnn1(x, (rnn1_hidden,
rnn1_cell))
        if self.training:
            rnn1_hidden = self.zoneout(rnn1_hidden, rnn1_hidden_next)
        else:
            rnn1_hidden = rnn1_hidden_next
        x = x + rnn1_hidden

        # Compute second Residual RNN
        rnn2_hidden_next, rnn2_cell = self.res_rnn2(x, (rnn2_hidden,
rnn2_cell))
        if self.training:
            rnn2_hidden = self.zoneout(rnn2_hidden, rnn2_hidden_next)
        else:
            rnn2_hidden = rnn2_hidden_next
        x = x + rnn2_hidden

        # Project Mels
        mels = self.mel_proj(x)
        mels = mels.view(batch_size, self.n_mels, self.max_r)[:, :, :self.r]
        hidden_states = (attn_hidden, rnn1_hidden, rnn2_hidden)
        cell_states = (rnn1_cell, rnn2_cell)

        # Stop token prediction
        s = torch.cat((x, context_vec), dim=1)
        s = self.stop_proj(s)
        stop_tokens = torch.sigmoid(s)

        return mels, scores, hidden_states, cell_states, context_vec,
stop_tokens


class Tacotron(nn.Module):
    def __init__(self, embed_dims, num_chars, encoder_dims, decoder_dims,
n_mels,
                 fft_bins, postnet_dims, encoder_K, lstm_dims, postnet_K,
num_highways,
                 dropout, stop_threshold, speaker_embedding_size):
        super().__init__()
        self.n_mels = n_mels
        self.lstm_dims = lstm_dims
        self.encoder_dims = encoder_dims
        self.decoder_dims = decoder_dims
        self.speaker_embedding_size = speaker_embedding_size
        self.encoder = Encoder(embed_dims, num_chars, encoder_dims,
                               encoder_K, num_highways, dropout)
        self.encoder_proj = nn.Linear(encoder_dims + speaker_embedding_size,
decoder_dims, bias=False)
```

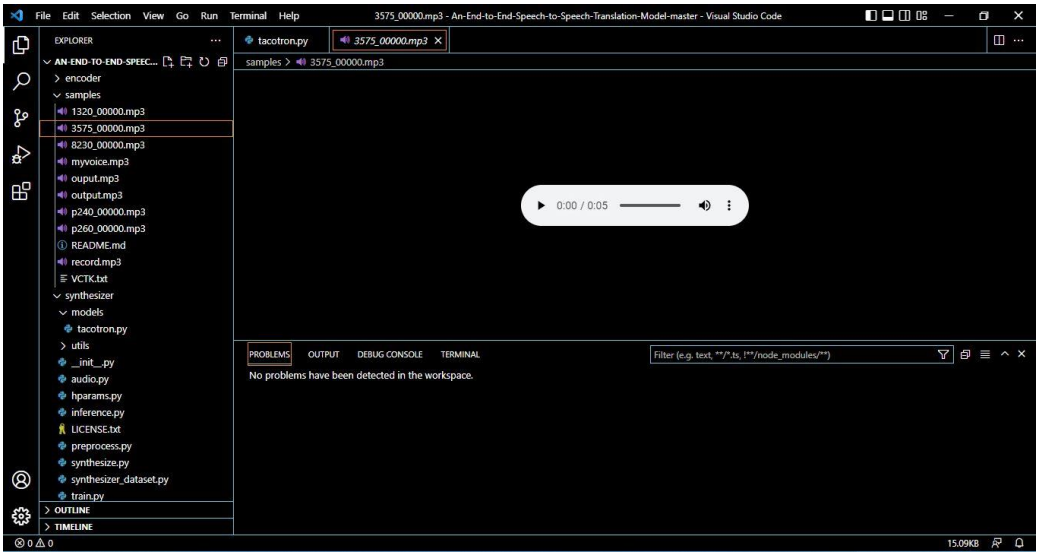
```
        self.decoder = Decoder(n_mels, encoder_dims, decoder_dims, lstm_dims,
                               dropout, speaker_embedding_size)
        self.postnet = CBHG(postnet_K, n_mels, postnet_dims,
                            [postnet_dims, fft_bins], num_highways)
        self.post_proj = nn.Linear(postnet_dims, fft_bins, bias=False)

        self.init_model()
        self.num_params()

        self.register_buffer("step", torch.zeros(1, dtype=torch.long))
        self.register_buffer("stop_threshold", torch.tensor(stop_threshold,
dtype=torch.float32))

    @property
    def r(self):
        return self.decoder.r.item()

    @r.setter
    def r(self, value):
        self.decoder.r = self.decoder.r.new_tensor(value, requires_grad=False)

    def forward(self, x, m, speaker_embedding):
        device = next(self.parameters()).device  # use same device as
parameters

        self.step += 1
        batch_size, _, steps  = m.size()

        # Initialise all hidden states and pack into tuple
        attn_hidden = torch.zeros(batch_size, self.decoder_dims,
device=device)
        rnn1_hidden = torch.zeros(batch_size, self.lstm_dims, device=device)
        rnn2_hidden = torch.zeros(batch_size, self.lstm_dims, device=device)
        hidden_states = (attn_hidden, rnn1_hidden, rnn2_hidden)

        # Initialise all lstm cell states and pack into tuple
        rnn1_cell = torch.zeros(batch_size, self.lstm_dims, device=device)
        rnn2_cell = torch.zeros(batch_size, self.lstm_dims, device=device)
        cell_states = (rnn1_cell, rnn2_cell)

        # <GO> Frame for start of decoder loop
        go_frame = torch.zeros(batch_size, self.n_mels, device=device)

        # Need an initial context vector
        context_vec = torch.zeros(batch_size, self.encoder_dims +
self.speaker_embedding_size, device=device)

        # SV2TTS: Run the encoder with the speaker embedding
```

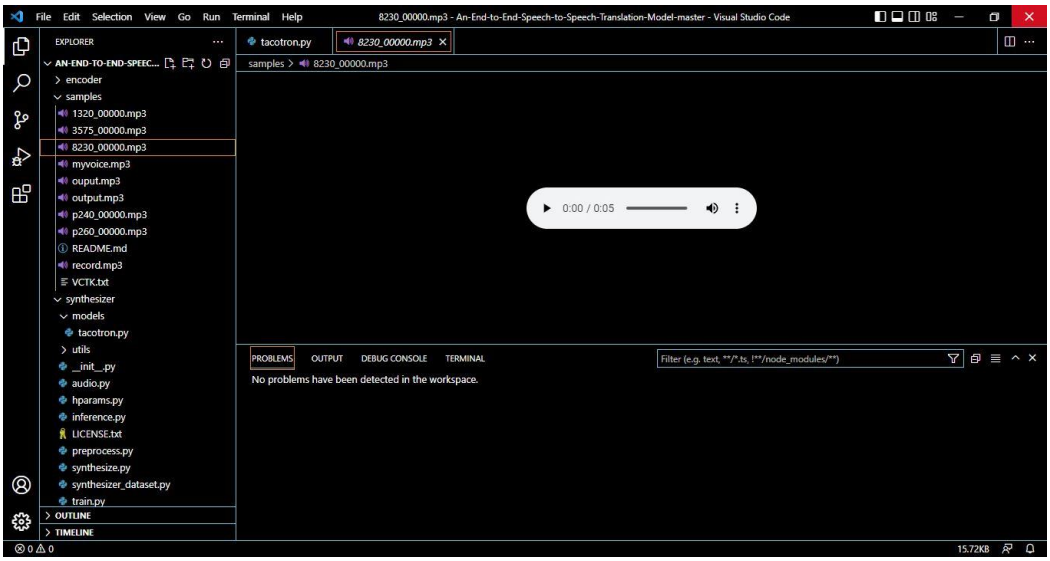
```
        # The projection avoids unnecessary matmuls in the decoder loop
        encoder_seq = self.encoder(x, speaker_embedding)
        encoder_seq_proj = self.encoder_proj(encoder_seq)

        # Need a couple of lists for outputs
        mel_outputs, attn_scores, stop_outputs = [], [], []

        # Run the decoder loop
        for t in range(0, steps, self.r):
            prenet_in = m[:, :, t - 1] if t > 0 else go_frame
            mel_frames, scores, hidden_states, cell_states, context_vec,
stop_tokens = \
                self.decoder(encoder_seq, encoder_seq_proj, prenet_in,
                             hidden_states, cell_states, context_vec, t, x)
            mel_outputs.append(mel_frames)
            attn_scores.append(scores)
            stop_outputs.extend([stop_tokens] * self.r)

        # Concat the mel outputs into sequence
        mel_outputs = torch.cat(mel_outputs, dim=2)

        # Post-Process for Linear Spectrograms
        postnet_out = self.postnet(mel_outputs)
        linear = self.post_proj(postnet_out)
        linear = linear.transpose(1, 2)

        # For easy visualisation
        attn_scores = torch.cat(attn_scores, 1)
        # attn_scores = attn_scores.cpu().data.numpy()
        stop_outputs = torch.cat(stop_outputs, 1)

        return mel_outputs, linear, attn_scores, stop_outputs

    def generate(self, x, speaker_embedding=None, steps=2000):
        self.eval()
        device = next(self.parameters()).device  # use same device as
parameters

        batch_size, _  = x.size()

        # Need to initialise all hidden states and pack into tuple for
tidyness
        attn_hidden = torch.zeros(batch_size, self.decoder_dims,
device=device)
        rnn1_hidden = torch.zeros(batch_size, self.lstm_dims, device=device)
        rnn2_hidden = torch.zeros(batch_size, self.lstm_dims, device=device)
        hidden_states = (attn_hidden, rnn1_hidden, rnn2_hidden)
```

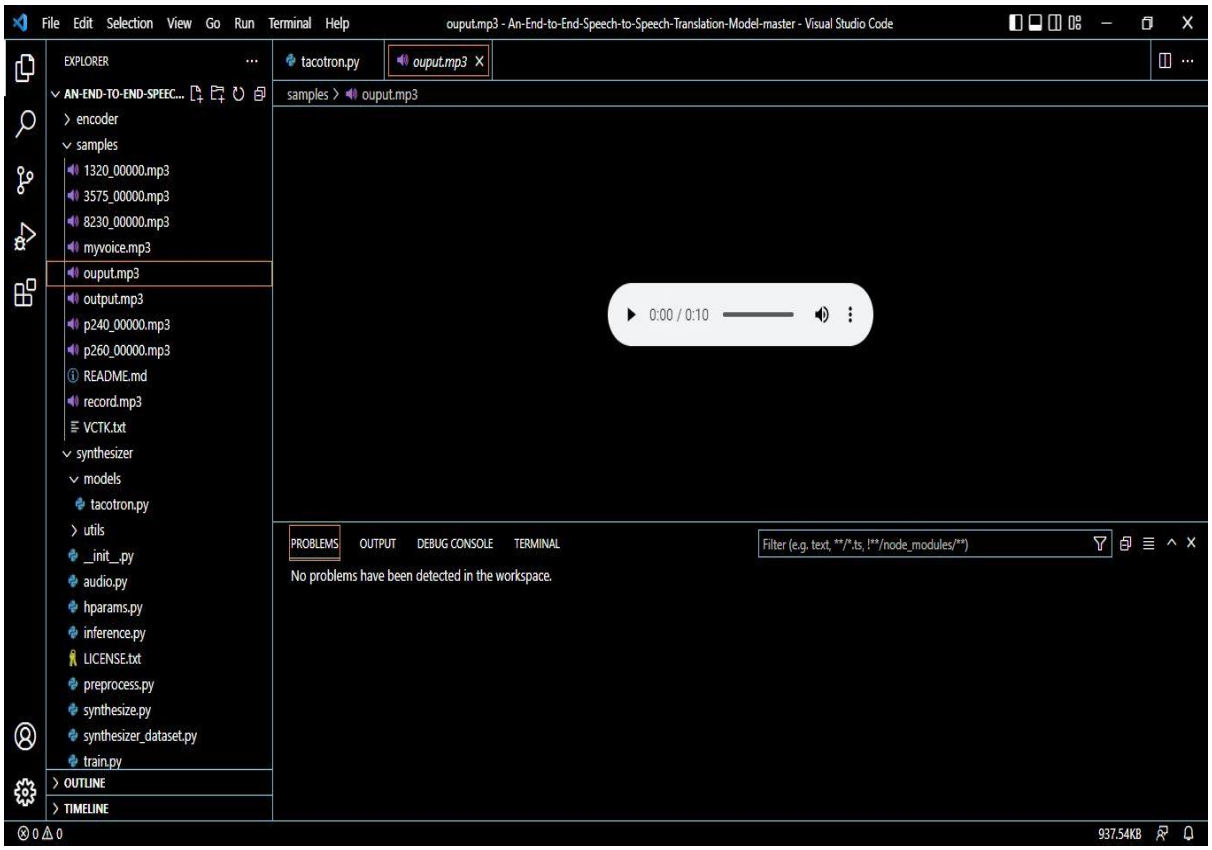
```
        # Need to initialise all lstm cell states and pack into tuple for
tidyness
        rnn1_cell = torch.zeros(batch_size, self.lstm_dims, device=device)
        rnn2_cell = torch.zeros(batch_size, self.lstm_dims, device=device)
        cell_states = (rnn1_cell, rnn2_cell)

        # Need a <GO> Frame for start of decoder loop
        go_frame = torch.zeros(batch_size, self.n_mels, device=device)

        # Need an initial context vector
        context_vec = torch.zeros(batch_size, self.encoder_dims +
self.speaker_embedding_size, device=device)

        # SV2TTS: Run the encoder with the speaker embedding
        # The projection avoids unnecessary matmuls in the decoder loop
        encoder_seq = self.encoder(x, speaker_embedding)
        encoder_seq_proj = self.encoder_proj(encoder_seq)

        # Need a couple of lists for outputs
        mel_outputs, attn_scores, stop_outputs = [], [], []

        # Run the decoder loop
        for t in range(0, steps, self.r):
            prenet_in = mel_outputs[-1][:, :, -1] if t > 0 else go_frame
            mel_frames, scores, hidden_states, cell_states, context_vec,
stop_tokens = \
            self.decoder(encoder_seq, encoder_seq_proj, prenet_in,
                         hidden_states, cell_states, context_vec, t, x)
            mel_outputs.append(mel_frames)
            attn_scores.append(scores)
            stop_outputs.extend([stop_tokens] * self.r)
            # Stop the loop when all stop tokens in batch exceed threshold
            if (stop_tokens > 0.5).all() and t > 10: break

        # Concat the mel outputs into sequence
        mel_outputs = torch.cat(mel_outputs, dim=2)

        # Post-Process for Linear Spectrograms
        postnet_out = self.postnet(mel_outputs)
        linear = self.post_proj(postnet_out)


        linear = linear.transpose(1, 2)

        # For easy visualisation
        attn_scores = torch.cat(attn_scores, 1)
        stop_outputs = torch.cat(stop_outputs, 1)

        self.train()
```

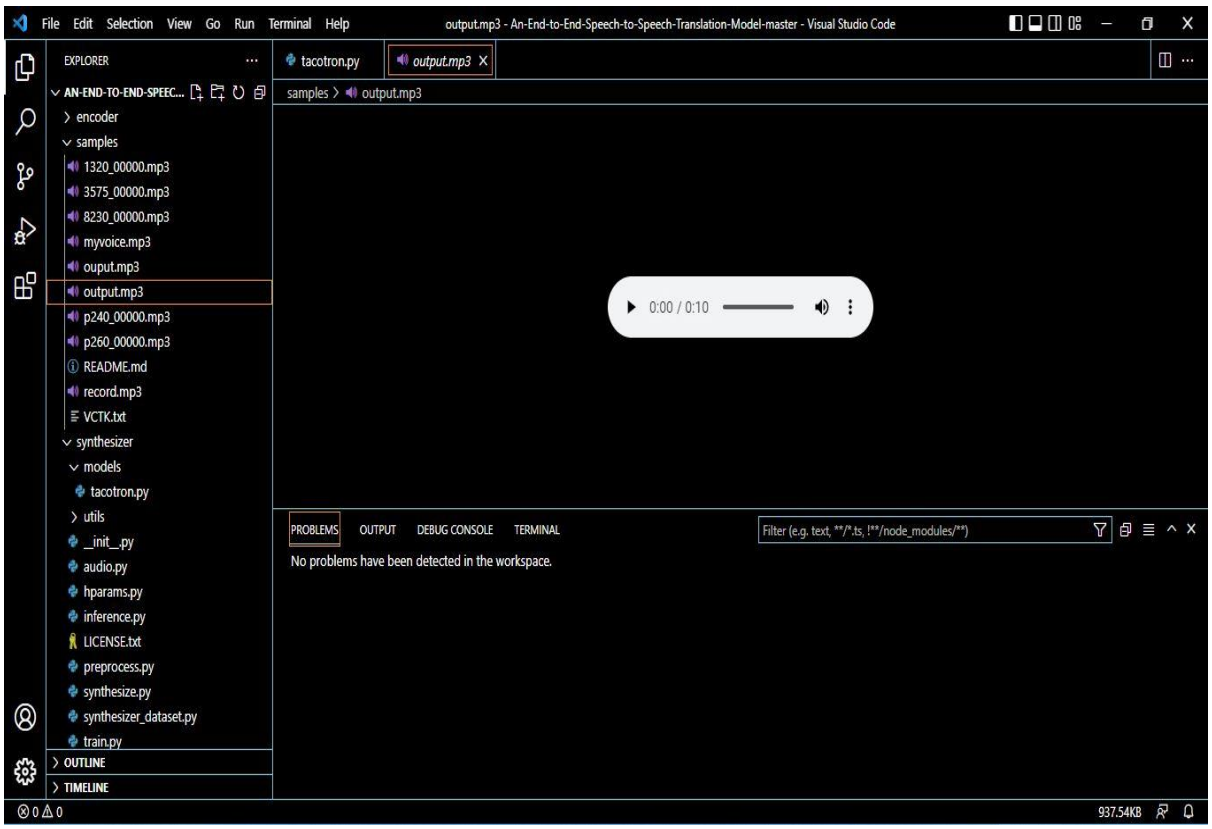
```
        return mel_outputs, linear, attn_scores

    def init_model(self):
        for p in self.parameters():
            if p.dim() > 1: nn.init.xavier_uniform_(p)

    def get_step(self):
        return self.step.data.item()

    def reset_step(self):
        # assignment to parameters or buffers is overloaded, updates internal
dict entry
        self.step = self.step.data.new_tensor(1)

    def log(self, path, msg):
        with open(path, "a") as f:
            print(msg, file=f)

    def load(self, path, optimizer=None):
        # Use device of model params as location for loaded state
        device = next(self.parameters()).device
        checkpoint = torch.load(str(path), map_location=device)
        self.load_state_dict(checkpoint["model_state"])

        if "optimizer_state" in checkpoint and optimizer is not None:
            optimizer.load_state_dict(checkpoint["optimizer_state"])

    def save(self, path, optimizer=None):
        if optimizer is not None:
            torch.save({
                "model_state": self.state_dict(),
                "optimizer_state": optimizer.state_dict(),
            }, str(path))
        else:
            torch.save({
                "model_state": self.state_dict(),
            }, str(path))


    def num_params(self, print_out=True):
        parameters = filter(lambda p: p.requires_grad, self.parameters())
        parameters = sum([np.prod(p.size()) for p in parameters]) / 1_000_000
        if print_out:
            print("Trainable Parameters: %.3fM" % parameters)
        return parameters
```


File Edit Selection View Go Run Terminal Help
1320_00000.mp3 - An-End-to-End-Speech-to-Speech-Translation-Model-master - Visual Studio Code
EXPLORER
AN-END-TO-END-SPEECH-TO-SPEECH-TRA...
encoder
samples
1320_00000.mp3
3575_00000.mp3
8230_00000.mp3
myvoice.mp3
ouput.mp3
output.mp3
p240_00000.mp3
p260_00000.mp3
README.md
record.mp3
VCTK.txt
synthesizer
models
tacotron.py
utils
__init__.py
audio.py
hparams.py
inference.py
LICENSE.txt
preprocess.py
synthesize.py
synthesizer_dataset.py
train.py
OUTLINE
TIMELINE
tacotron.py
1320_00000.mp3
samples > 1320_00000.mp3
0:00 / 0:05
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter (e.g. text, **/*.ts, !**/node_modules/**)
No problems have been detected in the workspace.
15.09KB


File Edit Selection View Go Run Terminal Help
3575_00000.mp3 - An-End-to-End-Speech-to-Speech-Translation-Model-master - Visual Studio Code
EXPLORER
AN-END-TO-END-SPEEC...
encoder
samples
1320_00000.mp3
3575_00000.mp3
8230_00000.mp3
myvoice.mp3
ouput.mp3
output.mp3
p240_00000.mp3
p260_00000.mp3
README.md
record.mp3
VCTK.txt
synthesizer
models
tacotron.py
utils
__init__.py
audio.py
hparams.py
inference.py
LICENSE.txt
preprocess.py
synthesize.py
synthesizer_dataset.py
train.py
OUTLINE
TIMELINE
tacotron.py
3575_00000.mp3
samples > 3575_00000.mp3
0:00 / 0:05
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter (e.g. text, **/*.ts, !**/node_modules/**)
No problems have been detected in the workspace.
15.09KB


File Edit Selection View Go Run Terminal Help
8230_00000.mp3 - An-End-to-End-Speech-to-Speech-Translation-Model-master - Visual Studio Code
EXPLORER
AN-END-TO-END-SPEEC...
encoder
samples
1320_00000.mp3
3575_00000.mp3
8230_00000.mp3
myvoice.mp3
ouput.mp3
output.mp3
p240_00000.mp3
p260_00000.mp3
README.md
record.mp3
VCTK.txt
synthesizer
models
tacotron.py
utils
__init__.py
audio.py
hparams.py
inference.py
LICENSE.txt
preprocess.py
synthesize.py
synthesizer_dataset.py
train.py
OUTLINE
TIMELINE
tacotron.py
8230_00000.mp3
samples > 8230_00000.mp3
0:00 / 0:05
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
Filter (e.g. text, **/*.ts, !**/node_modules/**)
No problems have been detected in the workspace.
15.72KB





کدها و صداها با نصب برنامه‌ها بدون خطا اجرا شد!