Malayalam Text-to Speech system

Guide: Jiby J P
Project Coordinator: Manilal D.L

Group 16
Kurian Benoy 34
kurianbenoy@mec.ac.in

Objective

- To build a Text to Speech(TTS) system in Malayalam
- Obtain the state of art result

- Module1 : EDA, dataset collection
- Module2: Train first TTS system in Malayalam
- Module3: Fine tune TTS system
- Module4: User Interface

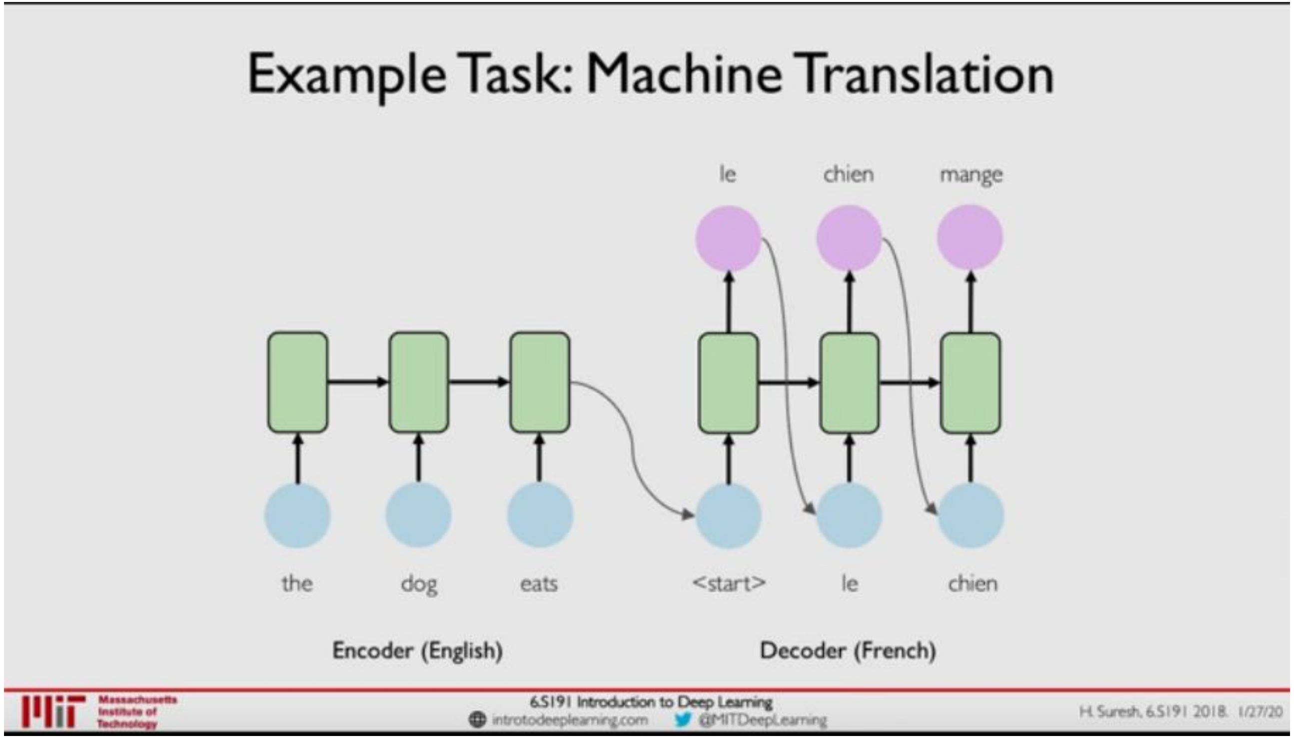
Work Done

Tactron2 paper Implementation

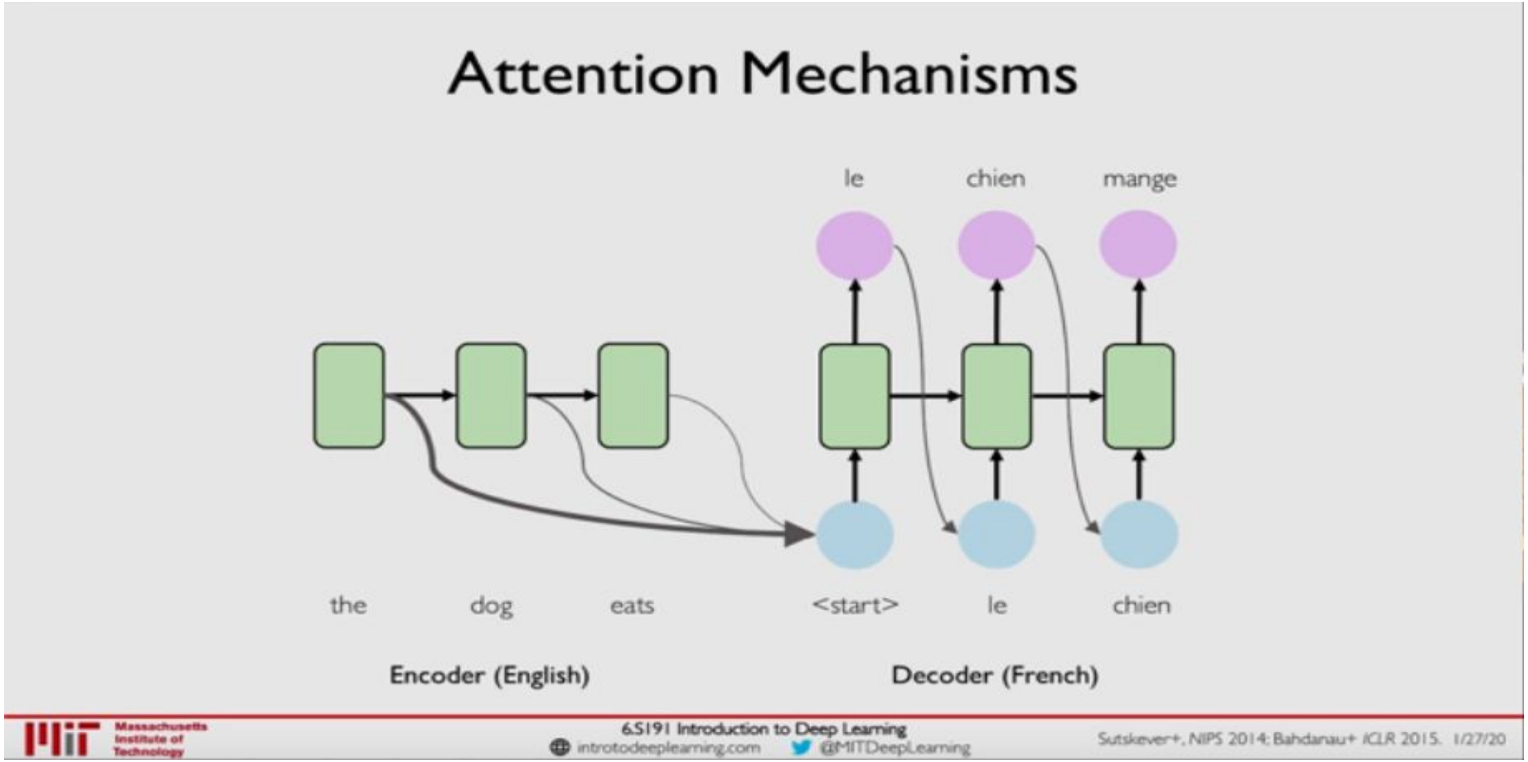
Tactron2, a neural network architecture for speech synthesis directly from text. The system consists of recurrent sequence to sequence feature extraction networks that map character embeddings to mel-scale spectrogram, which is currently one of the best Text to speech architectures, followed by modified Wavenet model acting as a synthesis domain. Model achieves 4.53 MOS score. Seq-2-seq models made of RNNs act as backbone networks for this.

Aim: To understand the paper and implement architecture

Sequence to Sequence models

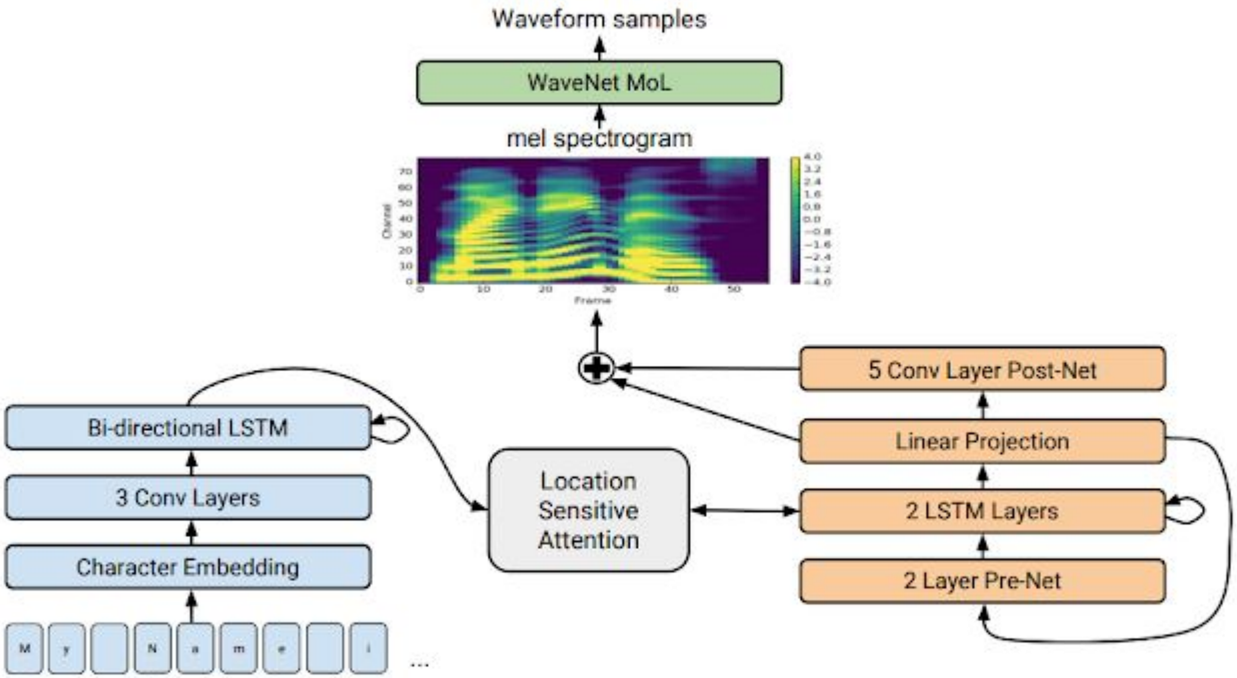


Sequence to Sequence models



Components of Tactron2 architecture

1. Location Layer
2. Encoder
3. Decoder
4. Attention Model
5. Posnet
6.



A detailed look at Tacotron 2's model architecture. The lower half of the image describes the sequence-to-sequence mod
that maps a sequence of letters to a spectrogram. For technical details, please refer to the paper.

Location Layer

In [7]:

```
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):
        processed_attention = self.location_conv(attention_weights_cat)
        processed_attention = processed_attention.transpose(1, 2)
        processed_attention = self.location_dense(processed_attention)
        return processed_attention
```

Attention Module

In [8]:

```
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):

        processed_query = self.query_layer(query.unsqueeze(1))
        processed_attention_weights = self.location_layer(attention_weights_cat)
        energies = self.v(torch.tanh(
            processed_query + processed_attention_weights + processed_memory))

        energies = energies.squeeze(-1)
        return energies

    def forward(self, attention_hidden_state, memory, processed_memory,
                attention_weights_cat, mask):
        alignment = self.get_alignment_energies(
            attention_hidden_state, processed_memory, attention_weights_cat)

        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
```

PostNet Module

```
class Postnet(nn.Module):
    """Postnet
        - Five 1-d convolution with 512 channels and kernel size 5
    """

    def __init__(self, hparams):
        super(Postnet, self).__init__()
        self.convolutions = nn.ModuleList()

        self.convolutions.append(
            nn.Sequential(

                (hparams.n_mel_channels, hparams.postnet_embedding_dim,
                         kernel_size=hparams.postnet_kernel_size, stride=1,
                         padding=int((hparams.postnet_kernel_size - 1) / 2),
                         dilation=1, w_init_gain='tanh'),
                nn.BatchNorm1d(hparams.postnet_embedding_dim))
        )

        for i in range(1, hparams.postnet_n_convolutions - 1):
            self.convolutions.append(
                nn.Sequential(
                    ConvNorm(hparams.postnet_embedding_dim,
                             hparams.postnet_embedding_dim,
                             kernel_size=hparams.postnet_kernel_size, stride=1,
                             padding=int((hparams.postnet_kernel_size - 1) / 2),
                             dilation=1, w_init_gain='tanh'),
                    nn.BatchNorm1d(hparams.postnet_embedding_dim))
            )

        self.convolutions.append(
            nn.Sequential(
                ConvNorm(hparams.postnet_embedding_dim, hparams.n_mel_channels,
                         kernel_size=hparams.postnet_kernel_size, stride=1,
                         padding=int((hparams.postnet_kernel_size - 1) / 2),
                         dilation=1, w_init_gain='linear'),
                nn.BatchNorm1d(hparams.n_mel_channels))
            )

    def forward(self, x):
        for i in range(len(self.convolutions) - 1):
            x = F.dropout(torch.tanh(self.convolutions[i](x)), 0.5, self.training)
        x = F.dropout(self.convolutions[-1](x), 0.5, self.training)

        return x
```

Encoder Module

In [11]:

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

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

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

    def forward(self, x, input_lengths):
        for conv in self.convolutions:
            x = F.dropout(F.relu(conv(x)), 0.5, self.training)

        x = x.transpose(1, 2)

        # pytorch tensor are not reversible, hence the conversion
        input_lengths = input_lengths.cpu().numpy()
        x = nn.utils.rnn.pack_padded_sequence(
            x, input_lengths, batch_first=True)

        self.lstm.flatten_parameters()
        outputs, _ = self.lstm(x)

        outputs, _ = nn.utils.rnn.pad_packed_sequence(
            outputs, batch_first=True)

        return outputs

    def inference(self, x):
        for conv in self.convolutions:
            x = F.dropout(F.relu(conv(x)), 0.5, self.training)

        x = x.transpose(1, 2)

        self.lstm.flatten_parameters()
        outputs, _ = self.lstm(x)

        return outputs
```

Decoder

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

        self.prenet = Prenet(
            hparams.n_mel_channels * hparams.n_frames_per_step,
            [hparams.prenet_dim, hparams.prenet_dim])

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

        self.attention_layer = Attention(
            hparams.attention_rnn_dim, hparams.encoder_embedding_dim,
            hparams.attention_dim, hparams.attention_location_n_filters,
            hparams.attention_location_kernel_size)

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

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

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

    def get_go_frame(self, memory):
        B = memory.size(0)
        decoder_input = Variable(memory.data.new(
            B, self.n_mel_channels * self.n_frames_per_step).zero_())
        return decoder_input

    def initialize_decoder_states(self, memory, mask):
        B = memory.size(0)
        MAX_TIME = memory.size(1)
```

Decoder

```
        self.attention_hidden = Variable(memory.data.new(
            B, self.attention_rnn_dim).zero_())
        self.attention_cell = Variable(memory.data.new(
            B, self.attention_rnn_dim).zero_())

        self.decoder_hidden = Variable(memory.data.new(
            B, self.decoder_rnn_dim).zero_())
        self.decoder_cell = Variable(memory.data.new(
            B, self.decoder_rnn_dim).zero_())

        self.attention_weights = Variable(memory.data.new(
            B, MAX_TIME).zero_())
        self.attention_weights_cum = Variable(memory.data.new(
            B, MAX_TIME).zero_())
        self.attention_context = Variable(memory.data.new(
            B, self.encoder_embedding_dim).zero_())

        self.memory = memory
        self.processed_memory = self.attention_layer.memory_layer(memory)
        self.mask = mask

    def parse_decoder_inputs(self, decoder_inputs):
        # (B, n_mel_channels, T_out) -> (B, T_out, n_mel_channels)
        decoder_inputs = decoder_inputs.transpose(1, 2)
        decoder_inputs = decoder_inputs.view(
            decoder_inputs.size(0),
            int(decoder_inputs.size(1)/self.n_frames_per_step), -1)
        # (B, T_out, n_mel_channels) -> (T_out, B, n_mel_channels)
        decoder_inputs = decoder_inputs.transpose(0, 1)
        return decoder_inputs

    def parse_decoder_outputs(self, mel_outputs, gate_outputs, alignments):
        # (T_out, B) -> (B, T_out)
        alignments = torch.stack(alignments).transpose(0, 1)
        # (T_out, B) -> (B, T_out)
        gate_outputs = torch.stack(gate_outputs).transpose(0, 1)
        gate_outputs = gate_outputs.contiguous()
        # (T_out, B, n_mel_channels) -> (B, T_out, n_mel_channels)
        mel_outputs = torch.stack(mel_outputs).transpose(0, 1).contiguous()
        # decouple frames per step
        mel_outputs = mel_outputs.view(
            mel_outputs.size(0), -1, self.n_mel_channels)
        # (B, T_out, n_mel_channels) -> (B, n_mel_channels, T_out)
        mel_outputs = mel_outputs.transpose(1, 2)

        return mel_outputs, gate_outputs, alignments
```

Decoder

```python
    def decode(self, decoder_input):
        cell_input = torch.cat((decoder_input, self.attention_context), -1)
        self.attention_hidden, self.attention_cell = self.attention_rnn(
            cell_input, (self.attention_hidden, self.attention_cell))
        self.attention_hidden = F.dropout(
            self.attention_hidden, self.p_attention_dropout, self.training)

        attention_weights_cat = torch.cat(
            (self.attention_weights.unsqueeze(1),
             self.attention_weights_cum.unsqueeze(1)), dim=1)
        self.attention_context, self.attention_weights = self.attention_layer(
            self.attention_hidden, self.memory, self.processed_memory,
            attention_weights_cat, self.mask)

        self.attention_weights_cum += self.attention_weights
        decoder_input = torch.cat(
            (self.attention_hidden, self.attention_context), -1)
        self.decoder_hidden, self.decoder_cell = self.decoder_rnn(
            decoder_input, (self.decoder_hidden, self.decoder_cell))
        self.decoder_hidden = F.dropout(
            self.decoder_hidden, self.p_decoder_dropout, self.training)

        decoder_hidden_attention_context = torch.cat(
            (self.decoder_hidden, self.attention_context), dim=1)
        decoder_output = self.linear_projection(
            decoder_hidden_attention_context)

        gate_prediction = self.gate_layer(decoder_hidden_attention_context)
        return decoder_output, gate_prediction, self.attention_weights

    def forward(self, memory, decoder_inputs, memory_lengths):

        decoder_input = self.get_go_frame(memory).unsqueeze(0)
        decoder_inputs = self.parse_decoder_inputs(decoder_inputs)
        decoder_inputs = torch.cat((decoder_input, decoder_inputs), dim=0)
        decoder_inputs = self.prenet(decoder_inputs)

        self.initialize_decoder_states(
            memory, mask=~get_mask_from_lengths(memory_lengths))

        mel_outputs, gate_outputs, alignments = [], [], []
        while len(mel_outputs) < decoder_inputs.size(0) - 1:
            decoder_input = decoder_inputs[len(mel_outputs)]
            mel_output, gate_output, attention_weights = self.decode(
                decoder_input)
            mel_outputs += [mel_output.squeeze(1)]
            gate_outputs += [gate_output.squeeze(1)]
            alignments += [attention_weights]
```

Gluing all together(with Tactron)

```python
class Tacotron2(nn.Module):
    def __init__(self,
                 num_chars,
                 num_speakers,
                 r,
                 postnet_output_dim=80,
                 decoder_output_dim=80,
                 attn_type='original',
                 attn_win=False,
                 attn_norm="softmax",
                 prenet_type="original",
                 prenet_dropout=True,
                 forward_attn=False,
                 trans_agent=False,
                 forward_attn_mask=False,
                 location_attn=True,
                 attn_K=5,
                 separate_stopnet=True,
                 bidirectional_decoder=False):
        super(Tacotron2, self).__init__()
        self.postnet_output_dim = postnet_output_dim
        self.decoder_output_dim = decoder_output_dim
        self.n_frames_per_step = r
        self.bidirectional_decoder = bidirectional_decoder
        decoder_dim = 512 if num_speakers > 1 else 512
        encoder_dim = 512 if num_speakers > 1 else 512
        proj_speaker_dim = 80 if num_speakers > 1 else 0
        # embedding layer
        self.embedding = nn.Embedding(num_chars, 512, padding_idx=0)
        std = sqrt(2.0 / (num_chars + 512))
        val = sqrt(3.0) * std  # uniform bounds for std
        self.embedding.weight.data.uniform_(-val, val)
        if num_speakers > 1:
            self.speaker_embedding = nn.Embedding(num_speakers, 512)
            self.speaker_embedding.weight.data.normal_(0, 0.3)
            self.speaker_embeddings = None
            self.speaker_embeddings_projected = None
        self.encoder = Encoder(encoder_dim)
        self.decoder = Decoder(decoder_dim, self.decoder_output_dim, r, attn_type, attn_win,
                               attn_norm, prenet_type, prenet_dropout,
                               forward_attn, trans_agent, forward_attn_mask,
                               location_attn, attn_K, separate_stopnet, proj_speaker_dim)
        if self.bidirectional_decoder:
            self.decoder_backward = copy.deepcopy(self.decoder)
        self.postnet = Postnet(self.postnet_output_dim)

    def _init_states(self):
        self.speaker_embeddings = None
        self.speaker_embeddings_projected = None

    @staticmethod
    def shape_outputs(mel_outputs, mel_outputs_postnet, alignments):
        mel_outputs = mel_outputs.transpose(1, 2)
        mel_outputs_postnet = mel_outputs_postnet.transpose(1, 2)
        return mel_outputs, mel_outputs_postnet, alignments
```

Gluing all together(with Tactron)

```python
    def forward(self, text, text_lengths, mel_specs=None, speaker_ids=None):
        self._init_states()
        # compute mask for padding
        mask = sequence_mask(text_lengths).to(text.device)
        embedded_inputs = self.embedding(text).transpose(1, 2)
        encoder_outputs = self.encoder(embedded_inputs, text_lengths)
        encoder_outputs = self._add_speaker_embedding(encoder_outputs,
                                                      speaker_ids)
        decoder_outputs, alignments, stop_tokens = self.decoder(
            encoder_outputs, mel_specs, mask)
        postnet_outputs = self.postnet(decoder_outputs)
        postnet_outputs = decoder_outputs + postnet_outputs
        decoder_outputs, postnet_outputs, alignments = self.shape_outputs(
            decoder_outputs, postnet_outputs, alignments)
        if self.bidirectional_decoder:
            decoder_outputs_backward, alignments_backward = self._backward_inference(mel_specs, encoder_outp
            return decoder_outputs, postnet_outputs, alignments, stop_tokens, decoder_outputs_backward, alig
        return decoder_outputs, postnet_outputs, alignments, stop_tokens

    @torch.no_grad()
    def inference(self, text, speaker_ids=None):
        embedded_inputs = self.embedding(text).transpose(1, 2)
        encoder_outputs = self.encoder.inference(embedded_inputs)
        encoder_outputs = self._add_speaker_embedding(encoder_outputs,
                                                      speaker_ids)
        mel_outputs, alignments, stop_tokens = self.decoder.inference(
            encoder_outputs)
        mel_outputs_postnet = self.postnet(mel_outputs)
        mel_outputs_postnet = mel_outputs + mel_outputs_postnet
        mel_outputs, mel_outputs_postnet, alignments = self.shape_outputs(
            mel_outputs, mel_outputs_postnet, alignments)
        return mel_outputs, mel_outputs_postnet, alignments, stop_tokens

    def inference_truncated(self, text, speaker_ids=None):
        """
        Preserve model states for continuous inference
        """
        embedded_inputs = self.embedding(text).transpose(1, 2)
        encoder_outputs = self.encoder.inference_truncated(embedded_inputs)
        encoder_outputs = self._add_speaker_embedding(encoder_outputs,
                                                      speaker_ids)
        mel_outputs, alignments, stop_tokens = self.decoder.inference_truncated(
            encoder_outputs)
        mel_outputs_postnet = self.postnet(mel_outputs)
        mel_outputs_postnet = mel_outputs + mel_outputs_postnet
        mel_outputs, mel_outputs_postnet, alignments = self.shape_outputs(
            mel_outputs, mel_outputs_postnet, alignments)
        return mel_outputs, mel_outputs_postnet, alignments, stop_tokens

    def _backward_inference(self, mel_specs, encoder_outputs, mask):
        decoder_outputs_b, alignments_b, _ = self.decoder_backward(
            encoder_outputs, torch.flip(mel_specs, dims=(1,)), mask,
            self.speaker_embeddings_projected)
        decoder_outputs_b = decoder_outputs_b.transpose(1, 2)
        return decoder_outputs_b, alignments_b

    def _add_speaker_embedding(self, encoder_outputs, speaker_ids):
        if hasattr(self, "speaker_embedding") and speaker_ids is None:
            raise RuntimeError(" [!] Model has speaker embedding layer but speaker_id is not provided")
        if hasattr(self, "speaker_embedding") and speaker_ids is not None:
            speaker_embeddings = self.speaker_embedding(speaker_ids)

            speaker_embeddings.unsqueeze_(1)
            speaker_embeddings = speaker_embeddings.expand(encoder_outputs.size(0),
                                                           encoder_outputs.size(1),
                                                           -1)
            encoder_outputs = encoder_outputs + speaker_embeddings
        return encoder_outputs
```

Gluing all together(with Tactron)

```python
    def decode(self, decoder_input):
        cell_input = torch.cat((decoder_input, self.attention_context), -1)
        self.attention_hidden, self.attention_cell = self.attention_rnn(
            cell_input, (self.attention_hidden, self.attention_cell))
        self.attention_hidden = F.dropout(
            self.attention_hidden, self.p_attention_dropout, self.training)

        attention_weights_cat = torch.cat(
            (self.attention_weights.unsqueeze(1),
             self.attention_weights_cum.unsqueeze(1)), dim=1)
        self.attention_context, self.attention_weights = self.attention_layer(
            self.attention_hidden, self.memory, self.processed_memory,
            attention_weights_cat, self.mask)

        self.attention_weights_cum += self.attention_weights
        decoder_input = torch.cat(
            (self.attention_hidden, self.attention_context), -1)
        self.decoder_hidden, self.decoder_cell = self.decoder_rnn(
            decoder_input, (self.decoder_hidden, self.decoder_cell))
        self.decoder_hidden = F.dropout(
            self.decoder_hidden, self.p_decoder_dropout, self.training)

        decoder_hidden_attention_context = torch.cat(
            (self.decoder_hidden, self.attention_context), dim=1)
        decoder_output = self.linear_projection(
            decoder_hidden_attention_context)

        gate_prediction = self.gate_layer(decoder_hidden_attention_context)
        return decoder_output, gate_prediction, self.attention_weights

    def forward(self, memory, decoder_inputs, memory_lengths):

        decoder_input = self.get_go_frame(memory).unsqueeze(0)
        decoder_inputs = self.parse_decoder_inputs(decoder_inputs)
        decoder_inputs = torch.cat((decoder_input, decoder_inputs), dim=0)
        decoder_inputs = self.prenet(decoder_inputs)

        self.initialize_decoder_states(
            memory, mask=~get_mask_from_lengths(memory_lengths))

        mel_outputs, gate_outputs, alignments = [], [], []
        while len(mel_outputs) < decoder_inputs.size(0) - 1:
            decoder_input = decoder_inputs[len(mel_outputs)]
            mel_output, gate_output, attention_weights = self.decode(
                decoder_input)
            mel_outputs += [mel_output.squeeze(1)]
            gate_outputs += [gate_output.squeeze(1)]
            alignments += [attention_weights]
```

Thank you!