CSC413 PA3

Ruijie Sun

March 2020

Part 1

i)

```
[11] class MyGRUCell(nn.Module):
         def __init__(self, input_size, hidden_size):
             super(MyGRUCell, self).__init__()

             self.input_size = input_size
             self.hidden_size = hidden_size

             # ------------
             # FILL THIS IN
             # ------------
             ## Input linear layers
             self.Wiz = nn.Linear(input_size, hidden_size)
             self.Wir = nn.Linear(input_size, hidden_size)
             self.Wih = nn.Linear(input_size, hidden_size)

             ## Hidden linear layers
             self.Whz = nn.Linear(hidden_size, hidden_size)
             self.Whr = nn.Linear(hidden_size, hidden_size)
             self.Whh = nn.Linear(hidden_size, hidden_size)


         def forward(self, x, h_prev):
             """Forward pass of the GRU computation for one time step.

             Arguments
                 x: batch_size x input_size
                 h_prev: batch_size x hidden_size

             Returns:
                 h_new: batch_size x hidden_size
             """

             # ------------
             # FILL THIS IN
             # ------------
             z = F.sigmoid(self.Wiz(x) + self.Whz(h_prev))
             r = F.sigmoid(self.Wir(x) + self.Whr(h_prev))
             g = F.tanh(self.Wih(x) + r * self.Whh(h_prev))
             h_new = (1 - z) * g + z * h_prev
             return h_new
```

ii)
error mode 1: does not work well dealing consonant pairs like "sh"
input: shopping is fun
translated: oppingspay isway unnay

error mode 2: when a vowel letter is within word, translation is wrong.
input: his hair is smooth
translated: ishway airway isway otablyway

Part 2

1)
$\widetilde{\alpha_i}^{(t)} = f(Q_t, K_i) = W_2(max(0, W_1[Q_t; K_i] + b_1)) + b2$

$\alpha_i^{(t)} = softmax(\widetilde{\alpha_i}^{(t)})$

$c_t = \sum_{i=1}^{T} \alpha_i^{(t)} V_i$

2)

```
def forward(self, inputs, annotations, hidden_init):
    """Forward pass of the attention-based decoder RNN.

    Arguments:
        inputs: Input token indexes across a batch for all the time step. (batch_size x decoder_seq_len)
        annotations: The encoder hidden states for each step of the input.
                     sequence. (batch_size x seq_len x hidden_size)
        hidden_init: The final hidden states from the encoder, across a batch. (batch_size x hidden_size)

    Returns:
        output: Un-normalized scores for each token in the vocabulary, across a batch for all the decoding time steps. (batch_
        attentions: The stacked attention weights applied to the encoder annotations (batch_size x encoder_seq_len x decoder_s
    """

    batch_size, seq_len = inputs.size()
    embed = self.embedding(inputs)  # batch_size x seq_len x hidden_size

    hiddens = []
    attentions = []
    h_prev = hidden_init
    for i in range(seq_len):
        # ------------
        # FILL THIS IN - START
        # ------------
        embed_current = embed[:,i,:]  # Get the current time step, across the whole batch

        context, attention_weights = self.attention(h_prev, annotations, annotations)  # batch_size x 1 x hidden_size

        embed_and_context = torch.cat((embed_current, context.squeeze(1)), 1)  # batch_size x (2*hidden_size)

        h_prev = self.rnn(embed_and_context, h_prev)  # batch_size x hidden_size
        # ------------
        # FILL THIS IN - START
        # ------------

        hiddens.append(h_prev)
        attentions.append(attention_weights)

    hiddens = torch.stack(hiddens, dim=1) # batch_size x seq_len x hidden_size
    attentions = torch.cat(attentions, dim=2) # batch_size x seq_len x seq_len

    output = self.out(hiddens)  # batch_size x seq_len x vocab_size
    return output, attentions
```

Part 3 Scaled Dot Product Attention

ScaledDotProduct

```python
class ScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(ScaledDotAttention, self).__init__()

        self.hidden_size = hidden_size

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=1)
        self.scaling_factor = torch.rsqrt(torch.tensor(self.hidden_size, dtype= torch.float))

    def forward(self, queries, keys, values):
        """The forward pass of the scaled dot attention mechanism.

        Arguments:
            queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x (k) x hidde
            keys: The encoder hidden states for each step of the input sequence. (batch_size x se
            values: The encoder hidden states for each step of the input sequence. (batch_size x

        Returns:
            context: weighted average of the values (batch_size x k x hidden_size)
            attention_weights: Normalized attention weights for each encoder hidden state. (batch_

            The output must be a softmax weighting over the seq_len annotations.
        """

        # ------------
        # FILL THIS IN
        # ------------
        batch_size = queries.shape[0]
        q = self.Q(queries)
        k = self.K(keys)
        v = self.V(values)


        unnormalized_attention = torch.bmm(k, q) * self.scaling_factor

        attention_weights = self.softmax(unnormalized_attention)
        context = torch.bmm(attention_weights.transpose(1, 2), v)
        return context, attention_weights
```

CausalScaledDotProduct

```python
class CausalScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(CausalScaledDotAttention, self).__init__()

        self.hidden_size = hidden_size
        self.neg_inf = torch.tensor(-1e7)

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=1)
        self.scaling_factor = torch.rsqrt(torch.tensor(self.hidden_size, dtype= torch.float))

    def forward(self, queries, keys, values):
        """The forward pass of the scaled dot attention mechanism.

        Arguments:
            queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x (k) x hidden_si
            keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_le
            values: The encoder hidden states for each step of the input sequence. (batch_size x seq_

        Returns:
            context: weighted average of the values (batch_size x k x hidden_size)
            attention_weights: Normalized attention weights for each encoder hidden state. (batch_siz

            The output must be a softmax weighting over the seq_len annotations.
        """

        # ------------
        # FILL THIS IN
        # ------------
        batch_size = queries.shape[0]
        q = self.Q(queries)
        k = self.K(keys)
        v = self.V(values)

        q = torch.transpose(q, 1, 2)

        unnormalized_attention = torch.bmm(k, q) * self.scaling_factor

        mask = torch.ones(unnormalized_attention.size()).byte().cuda()

        unnormalized_attention = unnormalized_attention.masked_fill_(mask.tril()==0, self.neg_inf)

        attention_weights = self.softmax(unnormalized_attention)

        context = torch.bmm(attention_weights.transpose(1, 2) , v)
        return context, attention_weights
```

TransformerEncoder

```python
[ ] class TransformerEncoder(nn.Module):
        def __init__(self, vocab_size, hidden_size, num_layers, opts):
            super(TransformerEncoder, self).__init__()

            self.vocab_size = vocab_size
            self.hidden_size = hidden_size
            self.num_layers = num_layers
            self.opts = opts

            self.embedding = nn.Embedding(vocab_size, hidden_size)

            # IMPORTANT CORRECTION: NON-CAUSAL ATTENTION SHOULD HAVE BEEN
            # USED IN THE TRANSFORMER ENCODER.
            # NEW VERSION:
            self.self_attentions = nn.ModuleList([ScaledDotAttention(
                                        hidden_size=hidden_size,
                                     ) for i in range(self.num_layers)])
            # PREVIONS VERSION:
            # self.self_attentions = nn.ModuleList([CausalScaledDotAttention(
            #                             hidden_size=hidden_size,
            #                          ) for i in range(self.num_layers)])
            self.attention_mlps = nn.ModuleList([nn.Sequential(
                                        nn.Linear(hidden_size, hidden_size),
                                        nn.ReLU(),
                                     ) for i in range(self.num_layers)])

            self.positional_encodings = self.create_positional_encodings()

        def forward(self, inputs):
            """Forward pass of the encoder RNN.

            Arguments:
                inputs: Input token indexes across a batch for all time steps in the sequence. (batch_size x seq_len)

            Returns:
                annotations: The hidden states computed at each step of the input sequence. (batch_size x seq_len x hidden_size)
                hidden: The final hidden state of the encoder, for each sequence in a batch. (batch_size x hidden_size)
            """

            batch_size, seq_len = inputs.size()
            # ------------
            # FILL THIS IN - START
            # ------------
            encoded = self.embedding(inputs)  # batch_size x seq_len x hidden_size

            # Add positinal embeddings from self.create_positional_encodings. (a'la https://arxiv.org/pdf/1706.03762.pdf, section 3.5)
            encoded = encoded + self.positional_encodings[:seq_len]

            annotations = encoded

            for i in range(self.num_layers):
              new_annotations, self_attention_weights = self.self_attentions[i](annotations, annotations, annotations)  # batch_size x se
              residual_annotations = annotations + new_annotations
              new_annotations = self.attention_mlps[i](residual_annotations)
              annotations = residual_annotations + new_annotations
            # ------------
            # FILL THIS IN - END
            # ------------

            # Transformer encoder does not have a last hidden layer.
            return annotations, None
```

TransformerDecoder

```
def forward(self, inputs, annotations, hidden_init):
    """Forward pass of the attention-based decoder RNN.

    Arguments:
        inputs: Input token indexes across a batch for all the time step. (batch_size x decoder_seq_len)
        annotations: The encoder hidden states for each step of the input.
                     sequence. (batch_size x seq_len x hidden_size)
        hidden_init: Not used in the transformer decoder
    Returns:
        output: Un-normalized scores for each token in the vocabulary, across a batch for all the decoding time steps. (batc
        attentions: The stacked attention weights applied to the encoder annotations (batch_size x encoder_seq_len x decoder_
    """

    batch_size, seq_len = inputs.size()
    embed = self.embedding(inputs)  # batch_size x seq_len x hidden_size

    # THIS LINE WAS ADDED AS A CORRECTION.
    embed = embed + self.positional_encodings[:seq_len]

    encoder_attention_weights_list = []
    self_attention_weights_list = []
    contexts = embed
    for i in range(self.num_layers):
      # ------------
      # FILL THIS IN - START
      # ------------
      new_contexts, self_attention_weights = self.self_attentions[i](embed, annotations, annotations)  # batch_size x seq_le
      residual_contexts = contexts + new_contexts
      new_contexts, encoder_attention_weights = self.encoder_attentions[i](residual_contexts, annotations, annotations)# bat
      residual_contexts = residual_contexts + new_contexts
      new_contexts = self.attention_mlps[i](residual_contexts)
      contexts = residual_contexts + new_contexts

      # ------------
      # FILL THIS IN - END
      # ------------

      encoder_attention_weights_list.append(encoder_attention_weights)
      self_attention_weights_list.append(self_attention_weights)

    output = self.out(contexts)
    encoder_attention_weights = torch.stack(encoder_attention_weights_list)
    self_attention_weights = torch.stack(self_attention_weights_list)

    return output, (encoder_attention_weights, self_attention_weights)
```

Question 5
Question 6

Part 4

Question 1 Instead of using ReLU, we can use tanh as activation function.

Question 3

```
[31] what_is("twelve minus fourteen")
negative
[32] what_is("twelve plus fourteen")
positive
[33] what_is("eight plus thousand")
positive
[34] what_is("eight minus thousand")
negative
[35] what_is("thousand minus eight")
positive
[36] what_is("eight minus thousand")
negative
[37] what_is("1 minus 14")
negative
[39] what_is("1 minus two")
positive
[40] what_is("one minus two")
negative
[41] what_is("three minus two minus eight")
negative
[42] what_is("three minus two")
negative
[43] what_is("one minus one minus one")
positive
[44] what_is("one minus one minus one plus ten")
positive
[45] what_is("one minus one plus ten minus one")
positive
[46] what_is("minus three plus eight")
positive
```

1. zero result are often missing.
2. 3 term calculation does not work well